
Pere László

A GNU/Linux programozása grafikus felületen

Változat: 1.16

Készült a Blum Szoftver Mérnökség Kft. Támogatásával.



Blum Szoftver Mérnökség Kft.
Értéket teremtünk

www.blumsoft.com

(c) 2006–2007 Pere László

A szerző hozzájárul a jelen mű változatlan formában történő sokszorosításához, beleértve az elektronikus vagy nyomdai formában készült másolatok készítését és terjesztését is. A hozzájárulás nem terjed ki a mű módosított formában való terjesztésére, beleértve, de nem korlátozva a mű bővítésére és részletekben történő reprodukálására.

Tartalomjegyzék

1. Bevezetés	7
1.1. A példaprogramokról	8
2. Az első lépések	9
2.1. Programozás Unix környezetben	9
2.1.1. A használt alkalmazások	9
2.1.2. A munkamenet	11
2.1.3. Több képernyőelem elhelyezése	14
2.1.4. Tulajdonságok és események	16
2.1.5. A visszahívott függvények szerkesztése	18
2.2. Programozás Windows környezetben	20
2.2.1. A MinGW környezet	20
2.3. Az objektumorientált programozás	24
2.3.1. Öröklődés és többalakúság	24
2.3.2. A típuskényszerítés	25
2.3.3. Az objektumtulajdonságok	26
2.3.4. Az általános típusú változó	28
2.4. Kapcsolat a képernyőelemek közt	29
3. Egyszerű felhasználói felületek készítése	33
3.1. Egyszerű képernyőelemek	34
3.1.1. A címke	35
3.1.2. A kép	37
3.1.3. A nyomógomb	38
3.1.4. A kapcsológomb	41
3.1.5. A beviteli mező	43
3.1.6. A forgatógomb	45
3.1.7. A kombinált doboz	46
3.1.8. A jelölőnégyzet	50
3.1.9. A rádiógomb	52
3.1.10A menüsáv	52
3.1.11A felbukkanó menü	56

4

3.1.12Az eszközsáv	60
3.2. Doboz jellegű elemek	60
3.2.1. Az ablak	61
3.2.2. A függőleges doboz	63
3.2.3. A vízszintes doboz	64
3.2.4. A táblázat	64
3.2.5. A keret	64
3.3. Általános eszközök	65
3.3.1. Az objektum	65
3.3.2. Az általános képernyőelem	68
4. A G programkönyvtár	73
4.1. A G programkönyvtár típusai	73
4.2. Az általános célú makrók	75
4.3. A hibakereséshez használható eszközök	76
4.4. A dinamikus memóriakezelés	79
4.5. A karakterláncok kezelése	80
4.5.1. A karakterláncok kezelése egyszerűen	81
4.5.2. Az Unicode karakterláncok	86
4.5.3. A magas szintű karakterlánc-kezelés	97
4.6. Az iterált adattípusok	101
4.6.1. A kétszeresen láncolt lista	102
4.6.2. A fa	109
4.6.3. A kiegyensúlyozott bináris keresőfa	110
4.7. Állománykezelés	114
4.7.1. Az állománynevek kezelése	115
4.8. A parancssori kapcsolók kezelése	117
5. Osztályok készítése	129
5.1. Egyszerű osztály készítése	129
5.2. Tulajdonságok rendelése az osztályhoz	142
5.3. Tagok elrejtése	146
6. XML állományok kezelése	149
6.1. Az XML állomány szerkezete	149
6.2. Az XML programkönyvtár egyszerű használata	152
6.2.1. A fordítás előtti beállítás	152
6.2.2. A fejállományok betöltése	153
6.2.3. A dokumentum létrehozása és mentése	153
6.2.4. A dokumentum betöltése	156
6.2.5. A dokumentum bővítése	157
6.2.6. A bejárás és a lekérdezés	160
6.2.7. A dokumentum módosítása	167
6.3. Az XPath eszköztár	170

6.3.1. Egyszerű XPath kifejezések kiértékelése	172
6.3.2. Az összetett eredményű XPath kifejezések	176
6.3.3. Az XPath műveleti jelek és függvények	177
7. Többablakos alkalmazások	179
7.1. Az ablakok megnyitása, bezárása	179
7.2. A függvényhívással létrehozható ablakok	182
7.2.1. Az üzenet-ablakok	183
7.2.2. Az állománynév-ablak	187
7.2.3. A betűtípus-ablak	191
7.2.4. A színválasztó-ablak	194
8. Összetett képernyőelemek	201
8.1. Az alkalmazás-ablak	201
8.1.1. Az elmozdítható eszközsáv	201
8.1.2. A fő képernyőelem	207
8.1.3. Az állapot sor	208
8.2. A szövegszerkesztő	209
8.2.1. A szöveg olvasása és írása	211
8.2.2. A szövegbejárók	216
8.2.3. A szövegjelek	226
8.2.4. A cetlik	231
8.2.5. A kurzor és a kijelölt szöveg	246
8.2.6. A szöveg módosításának figyelése	254
8.3. A raktárak	254
8.3.1. Az általános raktár	254
8.3.2. A lista szerkezetű raktár	262
8.3.3. A fa szerkezetű raktár	268
8.4. A cellarajzolók	270
8.4.1. A szöveges cellarajzó	272
8.4.2. A képmegjelenítő cellarajzó	279
8.4.3. A kapcsoló cellarajzó	281
8.4.4. A folyamatjelző cellarajzó	286
8.5. A cellaelrendezés	287
8.6. A kombinált doboz	293
8.7. A fa képernyőelem	298
8.7.1. Az oszlopok finomhangolása	301
8.7.2. A fa képernyőelem beállításai	308
8.7.3. A fa képernyőelem használata	311
8.7.4. A fa képernyőelem jelzései	322
8.8. Az ikonmező	327
8.8.1. Az ikonmező cellarajzói	338
8.8.2. Az ikonmező jelzései	338

6

9. Kiegészítő eszközök	341
9.1. A forrásprogram terjesztése	341
9.2. A programváltozatok követése	342
9.2.1. Különbbségi állományok	342
9.3. Új forrás-állományok	345
9.3.1. Ikonok elhelyezése a programcsomagban	346
9.4. Többnyelvű programok készítése	348

1. fejezet

Bevezetés

A grafikus felhasználói felület készítésére alkalmas programkönyvtárak általában igen sok, nem ritkán több ezer, esetleg több tízezer függvényt biztosítanak a programozó számára, amelyekhez sok adattípus, elemi és összetett adatszerkezet társul, tovább bonyolítva a programozó munkáját.

A nyelvi szerkezethez meglehetősen komoly jelentéstani réteg kapcsolódik. Bizonyos műveleteket csak bizonyos sorrendben lehet elvégezni, az egyes képernyőlemek közti rokonsági, öröklődési rendszert is figyelembe kell venni és így tovább.

Nyilvánvaló, hogy egy ilyen összetett nyelvi szerkezet használatához nem kell annak minden elemét tökéletesen ismerni. A programozó a munkája során felhasználhatja a teljes eszköztárat műszaki alapossággal leíró dokumentációt, kikeresheti azokat a kulcsszavakat, szabályokat, amelyeket nem tud segítség nélkül felidézni.

Egy dolgot azonban nem szabad szem elől tévesztenünk! A dokumentáció segíti a munkát, a hatékony programozáshoz azonban ismeretekre, tapasztalatra van szükségünk. Nem tárolhatunk minden ismeretet a könyvekben, a program írásához valamilyen szinten ismernünk kell az eszközöket, amelyeket igénybe kívánunk venni. A grafikus felhasználói felülettel támogatott alkalmazások készítése során a hatékony programozáshoz olyan mennyiségű nyelvi tudásra van szükségünk, amely akár beszélt nyelvek esetén is elegendő arra, hogy kifejezzük magunkat. Aki tehát hatékonyan akar ilyen alkalmazásokat készíteni, esetleg annyit kényszerül tanulni, hogy az beszélt nyelvek esetén már az alap- vagy középfokú nyelvvizsgálóhoz is elegendő volna.

1.1. A példaprogramokról

A grafikus felhasználói felületet használó programok általában sok sornyi C programkódot tartalmazó, viszonylag nagyméretű programok. A programozók különféle eszközökkel próbálnak meg úrrá lenni a bonyolultságon, különféle módszerekkel próbálják meg érthető méretű részekre szabdalni a programkódot. Az egyik előszeretettel használt módszer a **static** kulcsszó használata, amelyet a bemutatott példaprogramokban is használunk, mégpedig a következő formában:

```
1 static void
2 fill_store_with_filenames(GtkListStore *store)
3 {
4     ...
5 }
```

A függvények létrehozásakor a **static** kulcsszó azt jelzi, hogy a függvény csak az adott állomány fordítása során érhető el, a függvény hatóköre az állományra (fordítási egységre), korlátozódik. A példaprogramokban a hatókör ilyen formában történő korlátozásával jelöltük, hogy a függvény helyi, különleges feladatot lát el, ellentétben az általános feladatot ellátó, kiterjedt hatókörű függvényekkel, amelyek az alkalmazás távoli részei közt teremtenek kapcsolatot.

2. fejezet

Az első lépések

Ebben a fejezetben áttekintjük hogyan készíthetünk egyszerű grafikus alkalmazást a GTK+ programkönyvtár segítségével. Nyilvánvaló, hogy ezek az ismeretek nem elegendőek ahhoz, hogy összetett grafikus alkalmazásokat készítsünk, de – ahogyan látni fogjuk – egyszerűbb alkalmazásokat meglepően könnyen készíthetünk.

2.1. Programozás Unix környezetben

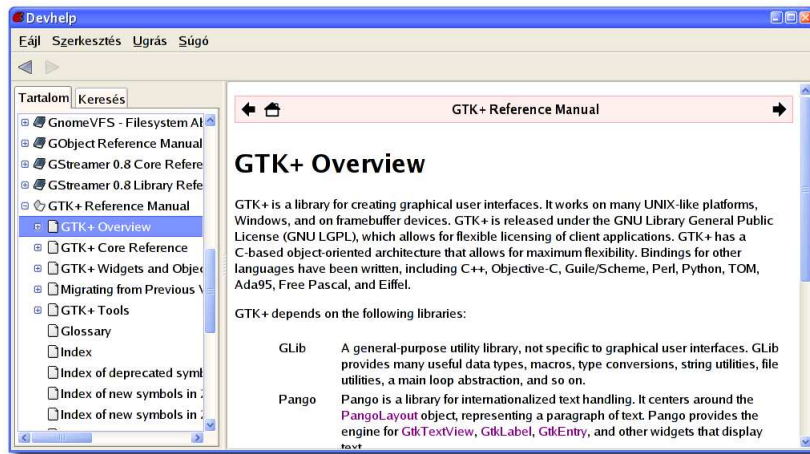
A GTK Unix környezetben természetes könnyedséggel használható, a szükséges alkalmazások és programkönyvtárak egyszerűen telepíthetők, az elkészített alkalmazás lefordítása általában nem jelent gondot. Mind ezen okok miatt érdemes valamilyen Unix környezetben megkezdenünk a munkát.

2.1.1. A használt alkalmazások

A könyvben erőteljesen támaszkodunk a Glade szerkesztőprogramra, ami lehetővé teszi, hogy a grafikus felhasználói felület elemeit felépítsük, hogy a programot „megrajzoljuk”. Grafikus felhasználói felületet a Glade nélkül is készíthetnénk, de tagadhatatlan, hogy a programmal egyszerűbben és gyorsabban készíthetjük el a programjainkat.

A GTK+ programkönyvtárról és az egyéb támogató programkönyvtárakról részletes fejlesztői leírás áll a rendelkezésünkre, amelyet legegyszerűbben a [devhelp](#) program segítségével érhetünk el. A [devhelp](#) programot láthatjuk a 2.1. ábrán.

Érdemes megemlíteni a [gtk-demo](#) programot, ami a GTK+ programkönyvtár néhány eszközét mutatja be. A programban egy időben figyelhetjük meg a példaprogram futását és a forrásprogramot, ami nagyszerű



2.1. ábra. A dokumentációt megjelenítő devhelp program

lehetőséget biztosít arra, hogy az egyszerűbb programozói fogásokat el-sajátíthassuk.

A programozás közben nyilvánvalóan szükségünk lesz egy szövegszer-kesztő programra, amellyel a programunk forrásállományait szerkeszt-jük. Természetesen minden programozó a jól bevált szövegszerkesztő programját részesíti előnyben, ügyelnünk kell azonban arra, hogy olyan szövegszerkesztőt válasszunk, amelyik képes érzékelni, ha a szerkesztett állomány megváltozott.

Bizonyos forrásállományokat ugyanis a Glade és a szövegszerkesztő program segítségével egyaránt és ügyelnünk kell, hogy e programok ne írják felül egymás munkáját. Szerencsés megoldás, ha a szövegszerkesz-tőben megnyitott összes állományt mentjük, mielőtt a Glade főablakában a forrásprogramok írását kérnénk, hogy a Glade a szövegszerkesztő ab-lakban látható legfrissebb programváltozatot módosítsa, majd, amikor a szövegszerkesztő jelzi, hogy a szerkesztett állományok egyike-másika megváltozott, egyszerűen kérjük az állomány újraolvasását. A **gvim** gra-fikus szövegszerkesztő például egy üzenetet jelenít meg, amikor a szöveg-szerkesztő ablakot kiválasztjuk, ha az állomány megváltozott. Az üzenet megjelenésekor kérhetjük a programot, hogy olvassa újra az állományt. Mivel a **gvim** az állomány újraolvasása után nem változtatja meg a kurzor helyét a szövegben, a programozási munkát igen kényelmesen folytathat-juk onnan, ahol megszakítottuk.

2.1.2. A munkamenet

A Glade szerkesztőprogram a `glade` vagy a `glade-2` parancs kiadásával indítható attól függően, hogy melyik változata van telepítve a számítógépünkre. Indítás után a program néhány egyszerű ablakot jelenít meg a képernyőn, amelyek a *főablak*, a *paletta* és a *tulajdonságok* nevet viselik.

A *főablak* (2.2. ábra) szokványos alkalmazásablak, menüivel, ikonjaival a szokásos módon végezhetjük a munkát, a középen található munkaterületen pedig a létrehozott grafikus elemeket választhatjuk ki.

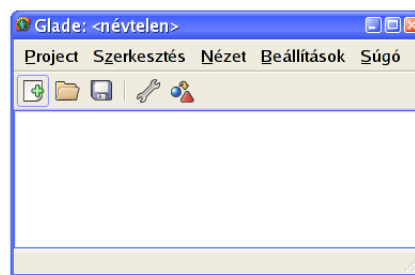
Kezdjük meg a munkát a *főablakban*! Válasszuk ki a *projekt* menü *új* menüpontját, hogy új programot készíthessünk. Ekkor a Glade megkérdezi, hogy GTK+ vagy GNOME programot kívánunk-e készíteni. Egyszerű programok esetében gyakorlatilag mindegy melyik lehetőséggel élünk – a GNOME néhány külön eszközt biztosít, de ezeket most még nem használjuk – szabadon dönthetünk.

Ha létrehoztuk az új projektet, válasszuk ki a *projekt* menü *beállítá-sok* menüpontját, hogy a programunk legfontosabb tulajdonságait be-állíthassuk. A *projekt beállítások* ablak (2.3. ábra) néhány eleme olyan információt tartalmaz, amire a későbbiekben szükségünk lesz.

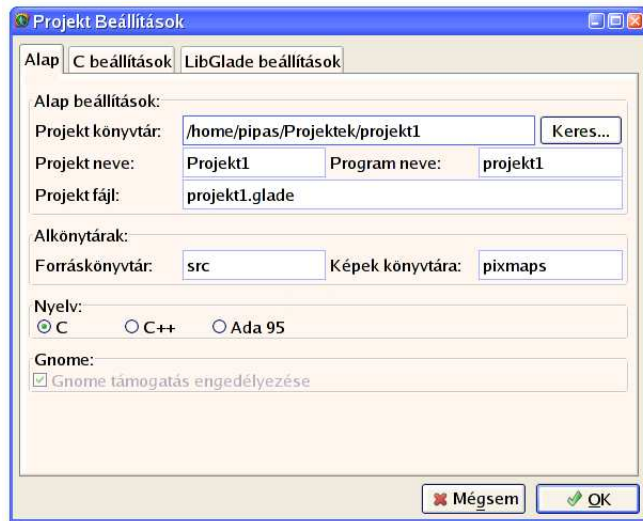
A *projekt könyvtár* mezőben megadhatjuk, hogy a programunk forrás-állományai melyik könyvtárba kerüljenek. Ez természetesen nem az a könyvtár, ahol a program a telepítés után elérhető, hanem az a könyvtár, ahol a fejlesztőmunkát végezzük.

A *program neve* mezőben megadhatjuk, hogy a programunknak mi lesz a neve, vagyis milyen programnév begépelésével indítható el maga a program.

A *projekt állomány* mezőben megadhatjuk, hogy a projekt könyvtáron belül melyik állományba kerüljenek a Glade számára elengedhetetlenül fontos adatok. Ez a projektállomány olyan formában tartalmazza a munkánkat, amit a Glade képes beolvasni és a későbbiekben szerkeszteni. A projekt állományra a programnak a futásakor nem feltétlenül van szüksége, ha azonban a program grafikus elemein a későbbiekben a Glade segítségével változtatni szeretnénk ezt az állományt kell megnyitnunk.



2.2. ábra. A Glade főablaka



2.3. ábra. A projekt legfontosabb beállításai

A **nyelv** mezőben megadhatjuk, hogy a Glade milyen programozási nyelven készítse el a programunk vázát. Amint láthatjuk a C, C++ és az Ada programozási nyelvek közül választhatunk, azaz a Glade ilyen nyelvű fejlesztésekben képes a segítségünkre lenni. E könyvben a C nyelvű programozással foglalkozunk, ez azonban természetesen nem jelenti azt, hogy a másik két programozási nyelv használata esetében győzőkeresen más módszereket kellene használnunk.

Tetszés szerint állítsunk be egy nevet az új programunk számára – ez a programfejlesztés egyik legnehezebb lépése –, majd az **ok** gomb lenyomása után vegyük szemügyre a **paletta** ablakot.

A **paletta** ablak (2.4. ábra) tartalmazza azokat a képernyőelemeket (*widget*), amelyeket a grafikus felhasználói felületen elhelyezhetünk.

A paletta nagy vonalakban a rajzolóprogramok esetében már megszokott módon működik. Az ablak bal felső sarkában található nyílra kattintva egy kijelölő eszközhöz jutunk, ami lehetővé teszi, hogy a már elhelyezett képernyőelemek kiválasszuk és módosítsuk, az ablak alsó részén látható ikonokra kattintva pedig kiválaszthatjuk, hogy milyen képernyőelemeket kívánunk elhelyezni a munkafelületen. Az elhelyezhető elemek csoportokba vannak rendezve. Megkülönböztetünk **GTK alap**, **GTK további** és **GNOME** elemeket. A csoportok közt található az **elavult** csoport, ami a működő, de új fejlesztésre nem ajánlott elemeket tartalmaz.

Mielőtt egyetlen elemet is elhelyeznénk, szükségünk van legalább egy ablakra, ami az elemeket tartalmazni fogja. A Glade többféle ablak létrehozását támogatja, kezdetben azonban kísérletezzünk a lehető legegyszerű-

rúbb, üres ablak létrehozásával! Kattintsunk a *paletta* ablak *GTK alap* eszközsávjának első, ablakot formázó ikonjára, ezzel hozunk létre egy új ablakot a képernyőn! Az ikonokra álva az egérkurzossal, egy kis segédablak (ún. *tooltip*) jelenik meg az ikon szöveges leírásával. Ez nagyon megkönnyíti a megfelelő elem kiválasztását.

Ha rákattintottunk az új ablak létrehozására szolgáló ikonra, azonnal megjelenik egy új szerkesztőablak a képernyőn (2.5. ábra), amelybe a képernyőelemeket elhelyezhetjük. Az új ablak neve egyben megjelenik a *főablak* középső területén, ahol a későbbiekben bármikor kiválaszthatjuk, hogy a tulajdonságait megváltoztassuk.

Helyezzünk most el egy nyomógombot az ablakban! Ezt igen egyszerűen megtehetjük: elég, ha a palettán belül kiválasztjuk a nyomógomb létrehozására szolgáló ikont, majd a szerkesztő ablak középső részére kattintva elhelyezzük a nyomógombot. Ezzel az egyszerű módszerrel egy ablakba csak egyetlen képernyőelemet helyezhetünk el, de a céljainknak most ez is megfelel.

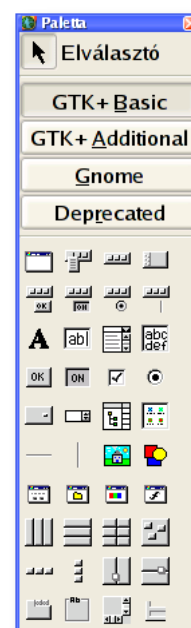
A szerkesztőablakban elhelyezett képernyőelemeket könnyedén tudjuk törölni is. Ehhez kattintsunk a *paletta* bal felső részén található nyíl alakú ikonra, majd kattintsunk a szerkesztőablak azon képernyőelemére, amelyet törölni szeretnénk. Ha kijelöltük a képernyőelemet – a sarkaiban megjelenő fekete négyzetek ezt jelzik – a  billentyű lenyomásával törölhető.

Ha elkészítettük az egyetlen nyomógombot tartalmazó ablakunkat mentjük a projektállományt. Ehhez válasszuk a *projekt* menü *mentés* menüpontját vagy kattintsunk az eszközsáv *mentés* ikonjára. A Glade nem ír külön üzenetet, ha a mentést sikeresen elvégezte.

Ezek után utasítanunk kell a programot, hogy hozza létre a projekthez tartozó programállományokat. Erre a műveletre a *projekt* menü *elkészítés* menüpontja vagy az eszközsáv *elkészítés* ikonja használható.

A következő feladatunk a program lefordítása. Ehhez először nyissunk egy parancssort, majd menjünk a projekt alapkönyvtárba, hogy a megfelelő parancsokat kényelmesen kiadhassuk!

Az első futtatandó parancs a `./autogen.sh`, ami a Glade által létreho-



2.4. ábra. A paletta



2.5. ábra. A szerkesztőablak

zott `autogen.sh` héjprogramot futtatja. Ez a program az `automake` és az `autoconf` programcsomagok segítségével létrehozza a fordítást vezérlő állományokat és a `./configure` parancs kiadásával elvégzi a fordítás előtti beállítást. A későbbi fejlesztési munka során a `./autogen.sh` parancsot általában nem kell kiadnunk, de a `./configure` parancsot a fordításhoz minden számítógépen használnunk kell, ahol a telepítést el akarjuk végezni.



A következő feladat a `make` program futtatása, amelyet a `make` parancs kiadásával kérhetünk. A `make` program az aktuális könyvtárban található `Makefile` állományt használva lefordítja a programunkat. Ez nagyobb projekt esetén hosszadalmas is lehet.

A sikeres fordítás után a programunk elérhető a projekt alapkönyvtárában található `src/` alkönyvtárban. Ha kiadjuk a `make install` parancsot, a `make` telepíti a programunkat, ehhez azonban rendszergazdai jogokra van szükségünk.

Adjuk ki tehát a megfelelő parancsokat az állományok létrehozására, a program fordítására és futtassuk a kész programot (a példából a parancsok kiadásának hatására megjelenő néhány sort eltávolítottuk helytakarékoság céljából):

```
$ ./autogen.sh
$ make
$ src/projekt1
```

Az utolsó parancssal elindított program a 2.6. ábrán látható. A program egyetlen nyomógombot tartalmazó ablakot jelenít meg és gyakorlatilag semmire nem használható. A nyomógomb lenyomásakor nem történik semmi, sőt az ablak bezárásakor a program tovább fut. E programból csak a `Ctrl` + `C` billentyűkombináció lenyomásával lehet kilépni.

2.1.3. Több képernyőelem elhelyezése

A következőkben megvizsgáljuk hogyan lehet egy ablakon belül több képernyőelemet elhelyezni. Meglepő, de ez nem olyan egyszerű, mint ahogyan gondolnánk, a Glade – és a GTK+ programkönyvtár – ugyanis nem engedi meg, hogy az ablakon belül több elem helyezkedjen el, csak akkor, ha azok különleges elemekben, úgynevezett dobozokban vannak.

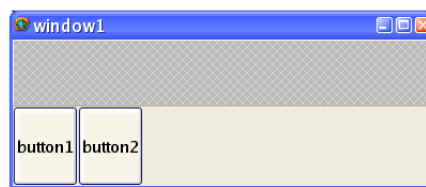
Ha kiléptünk volna a Glade programból indítsuk el újra és nyissuk meg a projektet a *projekt* menü *megnyitás* menüpontjával. A megnyitás után folytathatjuk a grafikus felhasználói felület szerkesztését.

Elsőként töröljük az ablakban elhelyezett nyomógombot úgy, hogy azt közben kimásoljuk a vágólapra! Jelöljük ki a nyomógombot, majd válasszuk a *szerkesztés* menü *kivágás* menüpontját. Ekkor a nyomógomb

eltűnik a szerkesztőablakból, de a vágólapon megmarad, hogy később a *szerkesztés* menü *beillesztés* menüpontjával újra elérhető legyen.

Most helyezzünk el az ablakban egy két sort tartalmazó függőleges dobozt! Válasszuk ki a *palette* ablakban a függőleges dobozt, kattintsunk a szerkesztőablak közepére – megadva, hogy hová szeretnénk elhelyezni a függőleges dobozt –, majd a megjelenő ablakban állítsuk be, hogy két sorból álló függőleges dobozt szeretnénk létrehozni. Ha megadtuk hány sorból álljon a függőleges doboz, a szerkesztőablakot a Glade két részre osztva jeleníti meg, jelezve ezzel, hogy most már két képernyőelem számára van benne hely.

Most osszuk két egymás mellett található részre a szerkesztőablak alsó sorát! Ehhez válasszuk ki a *palette* vízszintes dobozt jelző ikonját és kattintsunk a szerkesztőablak középső részének alsó sorába. Ha a megjelenő ablakban megadjuk, hogy két oszlopot szeretnénk létrehozni, a Glade az alsó sort két részre osztva jelöli, hogy most már két képernyőelemnek van helye az ablak alsó sorában.



2.7. ábra. Részekre osztott szerkesztőablak

Ilyen módon a szerkesztőablakot tetszőleges számú részre oszthatjuk egymásbaágyazott vízszintes és függőleges dobozok elhelyezésével.

Helyezzük most vissza az előbbieken eltávolított nyomógombot az alsó sor jobb oldalon található részébe! Ehhez jelöljük ki a szerkesztőablak jobb alsó részét egy kattintással, majd válasszuk ki a *szerkesztés* menü *beillesztés* menüpontját. Az előbb eltávolított nyomógomb újra megjelenik, de most a szerkesztőablaknak csak egy részét foglalja el.

Helyezzünk el a visszamásolt nyomógomb mellett egy új nyomógombot! Ehhez válasszuk ki a nyomógomb ikont a *palette* ablakból és kattintsunk a szerkesztőablak bal alsó részére. A megjelenő nyomógomb a kettes számot viseli, hogy meg tudjuk különböztetni a két egyforma képernyőelemet. Ezt láthatjuk a 2.7. ábrán.

Most helyezzünk el egy tetszőleges képernyőelemet a szerkesztőablak felső területén található részre úgy, hogy kiválassztjuk a megfelelő ikont, majd a szerkesztőablak adott területére kattintunk. Elhelyezhetünk például egy címkét, amely egy megváltoztathatatlan szöveget tartalmaz.



2.8. ábra.

A projekthez tartozó állományokat újra létre kell hoznunk, ha elkészültünk a felhasználói felület szerkesztéséhez. Ehhez először kattintsunk az eszközsáv *mentés* és *elkészítés* ikonjaira vagy válasszuk az ezeknek megfelelő menüpontokat a *projekt* menüből. Ha ezt megtettük, a módosított projekt elkészül,

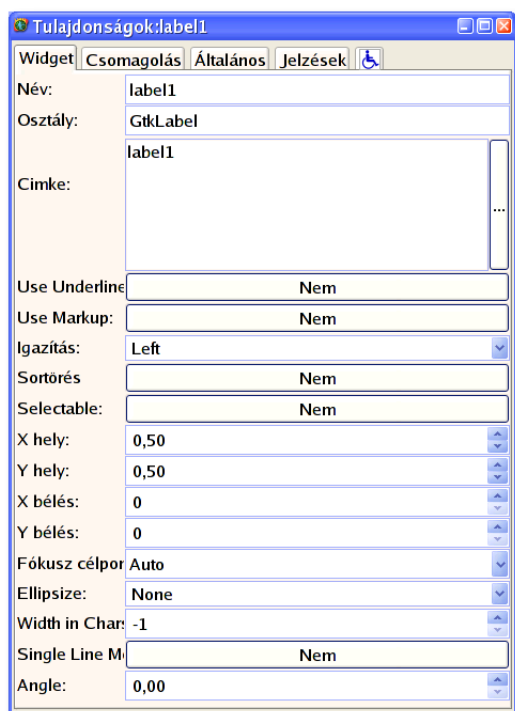
azt a `make` parancs kiadásával lefordíthatjuk és a már bemutatott módon telepíthetjük vagy helyben futtathatjuk.

A program új változatának képernyőképe a 2.8. ábrán látható. A program ezen változata az előző változathoz hasonlóan szintén nem használható semmire, még kilépni sem lehet belőle tisztességesen, viszont alapul szolgálhat egy következő, fejlettebb változat elkészítéséhez.

2.1.4. Tulajdonságok és események

A következőkben beállítjuk néhány képernyőelem alapvetően fontos tulajdonságait és eseményeket rendelünk az egyes nyomógombokhoz.

A képernyőelemek tulajdonságainak beállítására a *tulajdonságok* ablak használható, amely mindig az éppen kijelölt képernyőelem tulajdonságait mutatja. Az ablak a 2.9 ábrán látható. Itt a szerkesztőképernyőn elhelyezett címke legfontosabb tulajdonságait olvashatjuk le.



2.9. ábra. A címke tulajdonságai

Az ablak felső részén található *név* mezőben a képernyőelem neve látható. Ez az elkészített C nyelvű programban egy változó neve lesz, és erre a név megváltoztatásánál ügyelnünk kell. Nem adhatunk meg például ékezetes betűt tartalmazó nevet (bár a Glade ezt a problémát ügyesen megoldja az ékezetes betűk kicserélésével), hiszen a C programozási nyelvben ez nem megengedett.

Lejjebb található az *osztály* mező, ahol a képernyőelem típusára utaló kifejezést olvashatjuk¹. Ezt a mezőt nem lehet megváltoztatni.

A következő mező a *címke* nevet viseli. Ez tartalmazza azt a szöveget, ami a képernyőn megjelenik, ezt tehát szabadon megváltoztathatjuk és általában meg is kell változtatnunk. Mára a GTK+ könyvtár tökéletesen kezeli az Unicode szabványt, ezért a megfelelő nyelvi beállítások esetén nyugodtan használhatunk ékezetes karaktereket is a képernyőelemek tulajdonságainak megadásakor, viszont az is igaz,

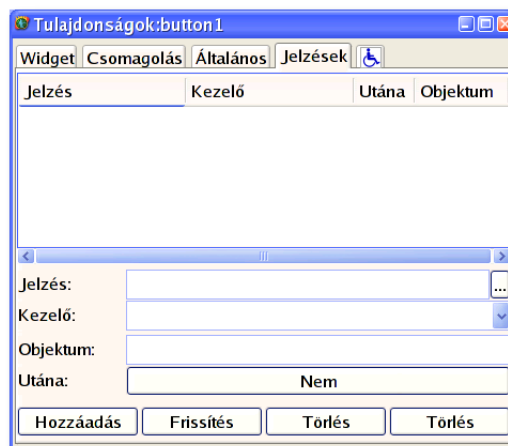
¹A GTK programkönyvtár objektumorientált programozási módszertant használ annak ellenére, hogy ezt a C programozási nyelv nem támogatja, ezért nevezzük osztálynak a képernyőelem típusát.

hogy a jó programokat angol nyelven írják és csak az elkészültük után fordítják le különböző nyelvekre az üzeneteket.

A 2.9. ábrán látható képernyőképhez hasonló adatok jelennek meg akkor is, ha a szerkesztőablakban a nyomógombra kattintunk. A *címke* mezőben átírva a szöveget természetesen a nyomógombon megjelenő szöveget módosíthatjuk.

Ha a képernyőelemek kinézetét beállítottuk, a következő lépésként az egyes képernyőelemekhez eseményeket rendelhetünk. Válasszuk ki valamelyik nyomógombot és a *tulajdonságok* ablak felső területén válasszuk ki a *jelzések* fület. Ekkor a 2.10. ábrán látható kép tárul a szemünk elé.

Az ábrán látható területen a jelzések kezelését állíthatjuk be. A GTK+ könyvtár a grafikus felhasználói felület történéseit eseményeknek (signal) nevezi. A felhasználói program beállíthatja, hogy az egyes események bekövetkeztekor mi történjen, vagyis pontosabban beállíthatja, hogy a GTK+ programkönyvtár az egyes események bekövetkeztekor az alkalmazás melyik függvényét hívja meg. Ezeket, a GTK+ által az események hatására hívott függvényeket visszahívott (callback) függvényeknek nevezzük. A 2.10. ábrán látható mezőkben az üzeneteket és a visszahívott függvényeket rendelhetjük egymáshoz.



2.10. ábra. A jelzések kezelése

Az ablak felső részén található táblázatban az egymáshoz rendelt jelzés–függvény párok közül válogathatunk, hogy módosítsuk őket. Ez a táblázat természetesen üres, ha még nem rendeltünk egyetlen jelzéshez sem visszahívott függvényt.

Az ablak alsó részében a *jelzés* mezőben választhatjuk ki melyik jelzéshez kívánunk függvényt rendelni, a *kezelő* mezőbe pedig a visszahívott függvény nevét írhatjuk be. Az *objektum* mezőbe egy változó nevét írhatjuk be, amelyet a hívás során a visszahívott függvény megkap. Ezt a mezőt egyszerűbb programok esetében üresen hagyhatjuk.

Ha a *jelzés*, *kezelő* és *objektum* mezőket kitöltöttük, az alsó részen található *hozzáadás* nyomógommbal az új jelzés–függvény párost elhelyezhetjük a táblázatban, amely a projekt adott képernyőeleme számára érvényes párosokat tartalmazza. Hasonló módon, a kijelölés után természetesen el is távolíthatjuk a táblázat egyes sorait a *törölés* nyomógommbal.

Készítsünk most visszahívott függvényeket a két nyomógommbhoz! Kat-

tintsunk a *tulajdonságok* ablak *jelzések* részén található *jelzés* mező melletti nyomógombra. Ekkor egy új ablak jelenik meg, amelyben kiválaszthatjuk melyik jelzés számára kívánunk visszahívott függvényt készíteni. Válasszuk ki a *clicked* jelzést. A nyomógombok esetében akkor keletkezik ez a jelzés, amikor a felhasználó a nyomógombra kattint.

Amint kiválasztottuk a *clicked* jelzést, a Glade a *kezelő* mezőbe beírja a javasolt függvénynevet a visszahívott függvény számára. Ezt a nevet természetesen megváltoztathatjuk, de erre inkább csak akkor van szükség, ha egyazon visszahívott függvényt akarunk használni több célra. Az egyszerűség kedvéért most fogadjuk el a felajánlott nevet és a *hozzáadás* nyomógomb segítségével rendeljük hozzá a jelzéshez a visszahívott függvényt. Természetesen szerencsés, ha a visszahívott függvény nevét megjegyezzük, bár ha logikusan neveztük el a képernyőelemeket és elfogadtuk a felajánlott függvénynevet, elég könnyű lesz felismerni a függvényt a programkód módosítása során.

Ezek után mentjük el a projektállományt és újra hozzuk létre az állományokat.

2.1.5. A visszahívott függvények szerkesztése

Az alapbeállítások esetében a Glade a projekt főkönyvtárában található *src/* alkönyvtárban helyezi el a *callbacks.c*, valamint a *callbacks.h* állományokat, amelyekben megtalálhatjuk a visszahívott függvényeket és a deklarációikat.

A Glade segítségével létrehozott visszahívott függvényeket ezekben az állományokban meg kell keresnünk és a függvények törzsét meg kell írunk, a Glade ugyanis csak az üres függvényeket helyezi el ezekben az állományokban.

Szerencsére a Glade elég fejlett, nem írja felül teljes egészében a *callbacks.c* és a *callbacks.h* állományokat akkor sem, ha a forrást újra létrehozzuk a grafikus felhasználói felület módosítása után. Az új függvényeket ilyenkor létrehozza, de az általunk módosított állományban nem tesz kárt.

1. példa. Vizsgáljuk most meg a *callbacks.h* állományt! A következő sorokat találjuk:

```

1  #include <gnome.h>
2
3
4  void
5  on_button3_clicked(GtkButton  *button,
6                      gpointer    user_data);
```

Amint látható az állomány csak egyetlen függvény, az általunk a Glade segítségével létrehozott visszahívott függvény típusát tartalmazza.

Nem minden visszahívott függvény típusa egyezik meg egymással, vannak egyszerűbb és vannak bonyolultabb argumentumokkal rendelkező visszahívott függvények. A Glade gondoskodik arról, hogy az adott jelzéshez a megfelelő típusú függvényt hozza létre.

2. példa. Vizsgáljuk meg most a `callbacks.c` állományt, amely a következő sorokat tartalmazza:

```

1  #ifdef HAVE_CONFIG_H
2  #   include <config.h>
3  #endif
4
5  #include <gnome.h>
6
7  #include "callbacks.h"
8  #include "interface.h"
9  #include "support.h"
10
11
12  void
13  on_button3_clicked(GtkButton *button,
14                    gpointer user_data)
15  {
16
17  }
```

Amint látható egyszerűen csak ki kell töltenünk a már előkészített függvényt a függvénytörzs megírásával. A Glade gondoskodott a megfelelő fejláncok betöltéséről és elkészítette a függvényt. Módosítsuk most az állományt, a következőképpen:

```

1  #ifdef HAVE_CONFIG_H
2  #   include <config.h>
3  #endif
4
5  #include <stdio.h>
6  #include <gnome.h>
7
8  #include "callbacks.h"
9  #include "interface.h"
10 #include "support.h"
```

```

11
12
13 void
14 on_button3_clicked(GtkButton *button,
15                     gpointer user_data)
16 {
17     printf("Igen\n");
18     exit(EXIT_SUCCESS);
19 }

```

Amint látjuk gondoskodtunk a `stdlib.h` betöltéséről és elhelyeztük a függvényben az `exit()` függvény hívását.

A `gtk_exit()` függvény nagyon hasonlít a C programkönyvtár `exit()` függvényére, azaz a programból való kilépésre használható. A GTK programkönyvtár újabb változatainak dokumentációja szerint a `gtk_exit()` függvény elavult, helyette az eredeti, szabványos `exit()` könyvtári függvényt javasolt használni.

2.2. Programozás Windows környezetben

2.2.1. A MinGW környezet

A MinGW (*Minimalist GNU for Windows*, kisméretű GNU Windows számára) fejlesztő és futtatókörnyezet a Cygwin környezetnél kisebb, a segítségével fordított alkalmazások jobban illeszkednek a Windows környezetbe. A MinGW környezet telepítése és karbantartása azonban nem olyan egyszerű, odafigyelést és némi hozzáértést igényel. A következő oldalakon a telepítéshez és a használathoz szükséges legfontosabb ismereteket mutatjuk be.

A telepítés első lépéseként az MSYS alaprendszer telepítését kell elvégeznünk. Ez a könyv megírásakor a `MSYS-1.0.10.exe` állomány [?] futtatásával végezhető el. A telepítés során a program alapértelmezés szerint a `C:\msys\1.0` könyvtárba másolja a szükséges állományokat és létrehoz egy indítóíkon a képernyőn.

Az MSYS rendszer ikonjával egy parancssoros felületet kapunk – ezt láthatjuk a 2.11. kép alsó, világos ablakaként –, amivel de túl sok program nem érhető el, hiszen a fejlesztők célja egy kimondottan egyszerű rendszer elkészítése volt.

Ha a telepített környezetben nem csak futtatni, hanem módosítani is akarjuk a programjainkat, akkor következő lépésként telepítsük az MSYS szoftverfejlesztő csomagját (*msys software development kit*), ami jelenleg az `msysDTK-1.0.1.exe` állomány [?] állomány futtatásával végezhető el.

A telepítés szintén a `C:\msys\1.0` könyvtárban történik, így az MSYS parancssorban válnak elérhetővé a fejlesztéshez szükséges programok.

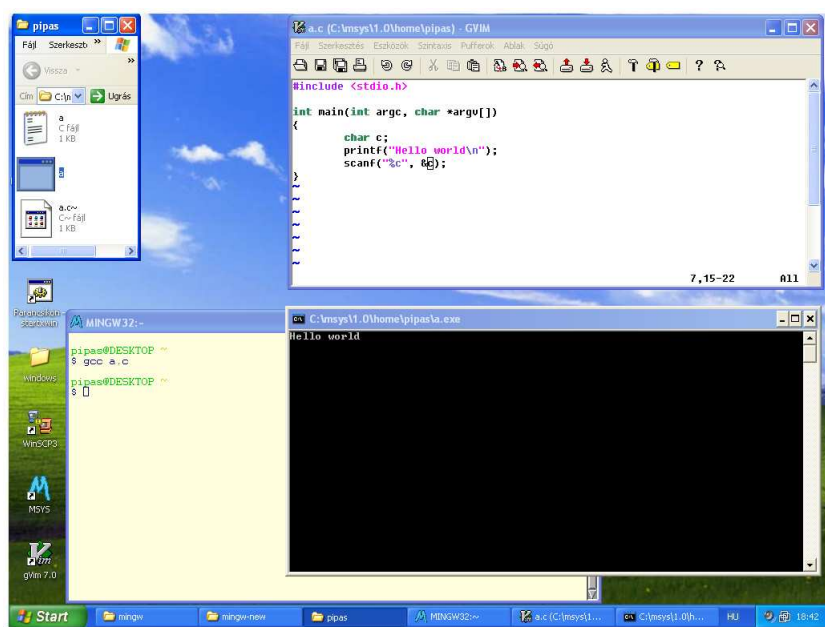
Az MSYS fejlesztőkörnyezet legfontosabb elemei az `autoconf`, az `automake`, a futtatásukhoz szükséges Perl rendszer és az `scp` ügyfél. Ezek a programok szükségesek lehetnek, ha a fejlesztés során például új forrásállományt szeretnénk az alkalmazáshoz adni, de az egyszerűbb módosítások elvégzéséhez nincs rájuk szükség. Az MSYS fejlesztőkörnyezet viszont nem tartalmaz C fordítót, ezért a következő lépés ennek a telepítése legyen.

Az MSYS alaprendszer telepítésekor létrejön a `C:\msys\1.0\mingw` könyvtár, ami a fordítóprogramokat, a programkönyvtárakat és a fejlálmányokat fogja tartalmazni. Ezt a könyvtárat az MSYS parancssorból `/mingw` néven érhetjük el, ami nem meglepő, hiszen az MSYS rendszer gyökérkönyvtára a `C:\msys\1.0`.

Mielőtt azonban leírnánk hogyan másolhatjuk be a fordítóprogramot és a szükséges programkönyvtárakat, érdemes megemlítenünk, hogy azoknak természetes helye a Windows könyvtárszerkezetben a `\mingw` könyvtár tetszőleges, általunk meghatározott meghajtó gyökérkönyvtárában. Ha azt akarjuk például, hogy a fordítóprogram és a könyvtárak a `C:\mingw` könyvtárban legyenek és ez a könyvtár az MSYS rendszerben a `/mingw/` könyvtárban jelenjen meg, akkor az MSYS parancssorban a `vim` szövegszerkesztőben el kell készítenünk a megfelelő `/etc/fstab` állományt. Ehhez az MSYS a `/etc/fstab.sample` példállománnyal járul hozzá, ami magától értetődővé teszi a munkát. Az egyszerűség érdekében megtehetjük azonban, hogy a fordítóprogramokat és a programkönyvtárakat egyszerűen a `C:\msys\1.0\mingw` könyvtárban csomagoljuk ki, ehhez ugyanis nem kell az `/etc/fstab` állományt módosítanunk.

A programfordításhoz szükséges állományok telepítéséhez egyszerűen ki kell csomagolnunk néhány állományt a `/mingw/` könyvtárba az MSYS `tar` és `gzip` programjaival. Fontos azonban, hogy ezeket az állományokat a `/mingw/` könyvtárba csomagoljuk ki, ne pedig az MSYS gyökérkönyvtárában, mert úgy nem működnek a programok.

Csomagoljuk ki az MSYS futtatókörnyezetet, ami jelenleg az `mingw-runtime-3.9.tar.gz` állományban [?] található. A C nyelvű programok fordításához szükségünk lesz az assemblerre és a szerkesztőprogramra is, amelyek jelenleg a `binutils-2.16.91-20060119-1.tar.gz` állományban [?] találhatók. Hasonlóképpen kell kicsomagolnunk a C fordítót, amit a könyv íráskor a `gcc-core-3.4.5-20060117-1.tar.gz` állomány [?] tartalmaz. Hasonló módon kell kicsomagolnunk a Windows programozó felület (*application programming interface*) elemeit elérhetővé tevő programkönyvtárakat és fejlálmányokat, amelyet jelenleg a `w32api-3.6.tar.gz` állományban van.



2.11. ábra. Egyszerű program futtatása MinGW környezetben

Ha idáig eljutottunk, akkor megpihenhetünk egy kicsit, hiszen ettől a pillanattól fogva képesek vagyunk a szabványos C programkönyvtár elemeit tartalmazó C programokat természetes Windows EXE állományokká fordítani és azokat szöveges ablakban le is futtathatjuk. Ha a Vim honlapjáról [?] letöltjük a szövegszerkesztő Windows változatát, akkor kényelmesen, külön ablakban szerkeszthetjük is a forrásprogramokat. Ezt láthatjuk a 2.11. ábrán. A kép bal alsó részén láthatjuk az MSYS parancssort és a `gvim` szövegszerkesztő indítására használható ikonokat, a képernyő jobb felső részén pedig a szövegszerkesztőben szerkesztett C programot. A kép alsó részén a világos színű MSYS parancsot és a lefűritott program futtatásakor megjelenő szöveges ablakot, ami sötét színnel jelenik meg.

Ha szükségünk van rá, ezek után hasonlóan egyszerű módon telepíthetjük fel a C++ fordítót ([gcc-g++-3.4.5-20060117-1.tar.gz](#)), az Ada fordítót ([gcc-ada-3.4.5-20060117-1.tar.gz](#)), a FORTRAN77 forítót ([gcc-g77-3.4.5-20060117-1.tar.gz](#)), a Java fordítót ([gcc-java-3.4.5-20060117-1.tar.gz](#)) és az Objective C fordítót ([gcc-objc-3.4.5-20060117-1.tar.gz](#)), amelyek a C fordítóval megegyező helyről [?] tölthetők le.

A telepítés következő lépéseiben azokat a programkönyvtárakat kell felmásolnunk, amelyek lehetővé teszik a grafikus felülettel ellátott programok lefordítását és futtatását. Mivel sokféle programkönyvtárra lesz szükségünk, ez a munka egy kissé időtrabló, lehangoló lehet, különösen azért, mert a szükséges programkönyvtárak listája a GTK és Gnome fejlesztése során változhat. Könnyen használható receptet nem tudunk tehát adni arra, hogy milyen állományokat töltsünk le és telepítsünk, néhány tanáccsal azonban segíthetjük a munkát.

Szerencsés, ha a telepítés előtt a Glade segítségével Linux alatt elkészítjük és kipróbálunk egy GTK projektet. Ha ezzel a különleges programkönyvtárat nem használó alkalmazáson próbáljuk ki a fordítást, könnyebb dolgunk lesz, mintha egy bonyolult alkalmazással próbálkoznánk.

Kezdjük a GTK programkönyvtár és a használatához szükséges programkönyvtárak telepítésével. Szerencsés, ha a programkönyvtárak `.zip` állományvégződéssel készített tömörített változatait töljtük le, ezeket ugyanis kevesebb munkával és gyorsabban tudjuk kitömöríteni, mintha az `.exe` változatok futtatásával végeznénk el a telepítést. Az állományokat abba a könyvtárba csomagoljuk ki, amelyik az MSYS rendszeren a `/mingw/` néven elérhető és ahová a C fordítóprogramot is elhelyeztük.

Ne feledjük el, hogy a programok fordításához a programkönyvtárak fejlesztést támogató változatait is ki kell csomagolnunk. Ezeknek az állományoknak a nevében a `dev` rövidítés utal arra, hogy a futtatáshoz nem szükségesek, a fejlesztés során azonban elengedhetetlen a használatuk.

Mindenképpen szükségünk lesz a G programkönyvtárra (ami je-

lenleg a [glib-2.12.1.zip](#) és a [glib-dev-2.12.1.zip](#) állományokban [?] található), a G programkönyvtárnak pedig szüksége van az [intl](#) programkönyvtárra (jelenleg a [gettext-0.14.5.zip](#) és a [gettext-dev-0.14.5.zip](#) állományokban [?] található).

2.3. Az objektumorientált programozás

A későbbiekben (a 73. oldalon található 4.1. szakaszban) részletesen bemutatjuk milyen fontosabb típusokat vezet be a G programkönyvtár és a könyv további részeiben azt is bemutatjuk milyen főbb típusai vannak a GTK+ programkönyvtárnak. Néhány fontos alapfogalmat és jellemző technikát azonban már a legegyszerűbb programok elkészítése előtt is érdemes megismernünk. Ezekről a fontos alapfogalmakról olvashatunk a következő néhány oldalon.

2.3.1. Öröklődés és többalakúság

A GTK+ programkönyvtár objektumorientált módszertan alapján készült, azt is mondhatnánk, hogy a GTK+ objektumorientált program.

Bizonyára sokan felkapják a fejüket erre a kijelentésre, hiszen tudják, hogy a GTK+ C programozási nyelven íródott és sokan hallották már, hogy a C programozási nyelv „nem objektumorientált”. Ez a kijelentés azonban csak egy kis helyesbítéssel igaz; azt mondhatjuk, hogy a C programozási nyelv nem támogatja az objektumorientált programozási módszertant, objektumorientált szemléletet. Attól azonban, hogy a nyelv nem támogatja még lehet objektumorientált programot írni C nyelven és pontosan ezt teszik a GTK+ alkotói.

Az objektumorientált programozási szemlélet lényeges eleme, hogy a programozó által létrehozott és használt objektumok osztályokba csoportosíthatók és az osztályok közti rokonsági kapcsolatok (az öröklődés) segítségével a program szerkezete egyszerűsíthető. Ennek megfelelően a GTK+ programkönyvtár által kezelt összes adattípus – köztük a képernyőelemeket leíró adattípusok –, jól átgondolt öröklődési rendszerbe vannak szervezve. A GTK+ által használt osztályok öröklődési rendszere a ???. oldalon található.

Az objektumorientált programozási módszertan egyik nagy előnye a többalakúság (polimorfizmus) lehetősége. A többalakúság azt jelenti, hogy a műveletek nem csak egy adott adatszerkezet esetében adnak helyes eredményt, többféle adatszerkezet kezelésére is használhatók. Általános szabályként elmondható, hogy az objektumorientált programozás esetében egy adott osztályhoz tartozó adattípuson (objektumon) minden művelet elvégezhető, amely az osztály szülőosztályain elvégezhető. Ennek a többalakúságnak a pontos megértését segíti a következő példa.

3. példa. A ?? . oldalon kezdődő öröklődési rendszer szerint a *GtkButton* osztály a *GtkWidget* osztály közvetett leszármazottja. Ez a többalakúság szabályai szerint azt jelenti, hogy minden művelet, amely elvégezhető a *GtkWidget* típussal, elvégezhető a *GtkButton* típussal is.

Ha tehát egy függvény *GtkWidget* típusú adatszerkezetet fogad, képes *GtkButton* adatszerkezetet is fogadni, így nincs szükség két külön függvényre.

Azok a programozási nyelvek amelyek támogatják az objektumorientált programozást nyilván kezelik a többalakúság kérdését. Nem így van ez azonban a C programozási nyelvben, amely nem támogatja az objektumorientált programozási szemléletet. A GTK+ alkotóinak megoldást kellett találniuk e problémára.

2.3.2. A típuskényszerítés

Az objektumorientált programozási módszertant támogató nyelvekben a többalakúság természetes része a nyelvnek, a C programozási nem áll ilyen eszköz a rendelkezésünkre. A G programkönyvtár alkotói a típuskényszerítés eszközével tették elérhetővé a többalakúságot a C programozási nyelvet használó programozók számára.

A G programkönyvtár minden osztályt a C nyelv struktúra típuskonstrukciós eszközével kezel, a többalakúságot pedig típuskényszerítést (*cast*, típus) végző makrókkal biztosítja és a GTK+ programkönyvtár is hasonló eszközöket használ. A GTK+ programkönyvtárban például minden képernyőelem típusához és minden osztályhoz azonos nevű makrók állnak a rendelkezésünkre a típuskényszerítéshez. A *GtkWidget* típushoz például a *GTK_WIDGET()* makró szolgál a típuskényszerítés és így a többalakúság megvalósításra. Látható, hogy a típusból úgy kapjuk meg a típuskényszerítő makró nevét, hogy a típus nevét csupa nagybetűvel írjuk, a szóhatárokat pedig aláhúzás karakterrel jelöljük.

4. példa. A 181. oldalon található 29. példa 34. sorában a *GtkButton* * típusú *button* nevű változóra az általánosabb *GtkWidget* * típusként van szükségünk, ezért használjuk a típuskényszerítésre szolgáló *GTK_WIDGET()* makró.

A típuskényszerítést végző makrók azonban nem csak egyszerűen megváltoztatják a mutatók típusát, de az öröklődési rendszernek – az osztályok gyermek-szülő viszonyainak – figyelembe vételével ellenőrzést is végeznek. A típuskényszerítés tehát a futási időben végzett típusellenőrzéssel együtt fejlett módon biztosítja a többalakúságot a programjaink számára.

2.3.3. Az objektumtulajdonságok

A G programkönyvtár az objektumok számára egy tulajdonság-rendszert biztosít, ami az objektumorientált programozás során használt öröklődési rendszert is figyelembe veszi. A programozó az egyes objektumokhoz névvel, típussal és értékkel rendelkező tulajdonságokat (*properties*) rendelhet, a programkönyvtár pedig egyszerűen használt eszközöket biztosít a tulajdonságok karbantartására.

A tulajdonságok kezelése során programkönyvtár függvényei figyelembe veszik az öröklődési rendszert. Ha egy objektum adott nevű tulajdonságát le akarjuk kérdezni, vagy meg akarjuk változtatni, akkor a programkönyvtár akkor is képes megtalálni az adott tulajdonságot, ha azt valamelyik szülőosztály vezette be és az objektum „csak” örökölte azt.



2.12. ábra. Forgatás

Mindazonáltal az objektumtulajdonságok kezelése kevésbé fontos, mint ahogyan azt első pillantásra gondolnánk és ennek több oka is van. A legtöbb alkalmazás elkészítéséhez a tulajdonságok alapértelmezett értéke teljesen megfelelő, felesleges azokat változtatni, ráadásul a fontos tulajdonságok lekérdezésére és megváltoztatására a GTK+ programkönyvtár külön függvényt biztosít, ami feleslegessé teszi a tulajdonságok közvetlen elérését.

Ráadásul a Glade program kényelmes és külön tanulás nélkül használható eszközöket biztosít a fontosabb tulajdonságok kezelésére, így leginkább csak akkor van szükségünk a tulajdonságok közvetlen elérésére, ha a kevésbé fontos tulajdonságokat a program futása során meg akarjuk változtatni. Ha például a címke szövegét nem vízszintesen akarjuk megjeleníteni (2.12. ábra), használhatjuk a Glade programot a beállításra, a forgatás szögének tulajdonságát csak akkor kell közvetlenül elérnünk, ha azt akarjuk, hogy a címke a program futása során forogjon.

Éppen azért azt javasoljuk az olvasónak, hogy ha gyors sikereket akar elérni, akkor egyszerűen ugorja át a könyv ezen szakaszát és csak akkor térjen ide vissza, ha megbizonyosodott arról, éppen ezekre az eszközökre van szüksége.

A G programkönyvtár az objektumok tulajdonságainak beállítására és lekérdezésére a következő két függvényt biztosítja.

```
void g_object_get(gpointer objektum, const gchar *név1,
mutatón1, ..., NULL);
```

A függvény segítségével az objektumok tulajdonságait kérdezhajjuk le a nevük alapján.

A függvény első paramétere mutató, az objektumot jelölő a memóriában.

A függvény további paraméterei tulajdonságneveket jelölő muta-

tók, amelyek után a megfelelő típusú érték hordozására használható memóriaterületeket jelölő mutatók. Ez utóbbi mutatók jelölik ki azokat a memóriaterületeket, ahová a függvény a tulajdonságok aktuális értékét elhelyezi.

A nevekből és mutatókból álló párok sorát – utolsó paraméterként – egy `NULL` értékkel kell lezárunk.

`void g_object_set(gpointer objektum, const gchar *név1, érték1, ..., NULL);` A függvény segítségével az objektumok tulajdonságait módosíthatjuk.

A függvény első paramétere mutató, amely a beállítani kívánt objektumot jelöli a memóriában.

A függvény további paramétere név/érték párokat adnak, ahol a név a megváltoztatandó tulajdonság nevét jelöli a memóriában, az érték pedig az adott tulajdonságnak megfelelő típusú érték.

A nevekből és értékekből álló párokat – utolsó paraméterként – `NULL` értékkel kell lezárunk.

A G programkönyvtár által támogatott tulajdonságok lekérdezését és beállítását mutatja be a következő két példa.

5. példa. A program futása során szeretnénk lekérdezni, hogy a beviteli mezőben a szövegkurzor hányadik karakternél áll. A következő sorok bemutatják hogyan tehetjük ezt meg.

```

1 void
2 on_entry001_move_cursor(GtkEntry      *entry,
3                           GtkMovementStep step,
4                           gint          count,
5                           gboolean      extend_selection,
6                           gpointer      user_data)
7 {
8     gint position;
9
10    g_object_get(entry,
11                 "cursor-position", &position, NULL);
12    g_message("%s(): %d", __func__, position);
13 }

```

A programrészlet egy visszahívott függvényt mutat be, amit a GTK+ programkönyvtár akkor hív, amikor a beviteli mezőben a szövegkurzort elmozdítjuk, a kurzormozgató billentyűk valamelyikével. A kurzor helyét természetesen máskor is lekérdezhetjük.

28

A függvény a 10–11. sorban hívja a `g_object_get()` függvényt, hogy a szövegkurzor pozícióját – ami egy egész érték – lekérdezze. A program ez után a 12. sorban kiírja a kapott értéket a szabványos kimenetre.

6. példa. Szeretnénk a program futásakor korlátozni egy beviteli mezőben a beírható, megjeleníthető karakterek számát. A következő néhány sor bemutatja hogyan tehetjük ezt meg.

```
1 void
2 on_entry002_realize(GtkWidget *widget,
3                     gpointer user_data)
4 {
5     g_object_set(widget, "max-length", 3, NULL);
6 }
```

A programrészlet egy visszahívott függvényt mutat be, amit a GTK+ programkönyvtár akkor hív, amikor a beviteli mező elkészült, de még nem jelent meg a képernyőn. Ez a tény nem befolyásolja a függvény működését, annak csak egy létező beviteli mezőre van szüksége ahhoz, hogy az adott tulajdonságok beállítsa.

A függvény az 5. sorban hívja a `g_object_set()` függvényt, hogy a karakterek számát korlátozó egész típusú tulajdonságot beállítsa. A beviteli mezők ezen tulajdonságát egyébként a Glade segítségével is beállíthatjuk, ha a program futása során állandó értéken akarjuk tartani.

2.3.4. Az általános típusú változó

A G programkönyvtár a `GValue` típust biztosítja különféle típusú értékek általános hordozójaként.

7. példa. A következő sorok bemutatják hogyan használhatjuk a `GValue` típust egy objektum tulajdonságainak lekérdezésére. (A program az 5. példa alacsony szintű eszközökkel készített változata.)

```
1 void
2 on_entry001_move_cursor(GtkEntry *entry,
3                           GtkMovementStep step,
4                           gint count,
5                           gboolean extend_selection,
6                           gpointer user_data)
7 {
8     GValue int_val = {0,};
9     gint position;
10 }
```

```

11  g_value_init(&int_val, G_TYPE_INT);
12  g_object_get_property(G_OBJECT(entry),
13      "cursor-position", &int_val);
14  position = g_value_get_int(&int_val);
15
16  g_message("%s(): %d", __func__, position);
17  }

```

8. példa. A következő példaprogram bemutatja hogyan állíthatjuk be a **GValue** típus segítségével egy objektum tulajdonságát. (A program a 6. példa alacsony szintű eszközökkel készített változata.)

```

1  void
2  on_entry002_realize(GtkWidget *widget,
3                      gpointer   user_data)
4  {
5      GValue int_val = {0,};
6
7      g_value_init(&int_val, G_TYPE_INT);
8      g_value_set_int(&int_val, 3);
9      g_object_set_property(G_OBJECT(widget),
10         "max-length", &int_val);
11  }

```

A példa 5. sorában egy **GValue** típusú változót hozunk létre, ami az objektumtulajdonság értékét hordozni fogja. Mivel a dokumentáció szerint **gint** típusú, a 7. sorban beállítjuk a **GValue** típusát **G_TYPE_INT** típusra, majd a 8. sorban a **g_value_set_int()** függvény segítségével beállítjuk az értéket is.

Az objektum tulajdonságának beállítása a 9-10. sorban látható, ahol a **g_object_set_property()** függvénnyel beállítjuk az adott nevű tulajdonság értékét. Figyeljük meg, hogy a többalakúságnak köszönhetően a beviteli mezőt a függvény a G programkönyvtár objektumaként (**G_OBJECT()**) kapja meg, mégis képes kideríteni, hogy tulajdonképpen beviteli mezőről van szó, amelynek van ilyen néven bejegyzett tulajdonsága.

2.4. Kapcsolat a képernyőelemek közt

Amikor a visszahívott függvényeket szerkesztjük szükségünk van az általunk készített felhasználói felületek egyéb elemeire is. Ha például a felhasználó rákattint az általunk készített ablak egy nyomógombjára, a visszahívott függvényben szükségünk van az adott ablak szövegbeviteli

30

mezőjének aktuális értékére, így meg kell találnunk magát a szövegbeviteli mezőt.

Erre a célra a Glade egy segédfüggvényt hoz létre, amely alapértelmezés szerint a projekt alapkönyvtárának `src/` alkönyvtárában található a `support.c` állományban. A függvény deklarációja ugyanebben a könyvtárban, a `support.h` állományban van. (A Glade mindkét állományt automatikusan felülírja, így módosítanunk ezeket nem szabad.)

```
GtkWidget *lookup_widget(GtkWidget *ind, const gchar
*név);
```

A függvény a képernyőelemek közt keres név szerint, lehetővé teszi, hogy az ablakon belüli képernyőelemeket megtaláljuk.

vi sszatérési érték A függvény vi sszatérési értéke a keresett képernyőelemet jelölő mutató, ha a keresés sikeres volt, `NULL`, ha nem.

ind A keresés kiindulópontja. A keresett elemnek a kiindulóponttal azonos ablakban kell lennie, különben a függvény nem találja meg.

név A keresett képernyőelem neve, amelyet a Glade tulajdonságok szerkesztésére alkalmas ablakában adtunk meg.

A függvény használatához a `support.h` állományt be kell töltenünk.

9. példa. Készítsünk most egy ablakot, amelyben egy nyomógomb és két beviteli mező található. A beviteli mezőknek adjuk az `entry1` és `entry2` nevet! Készítsünk vi sszahívott függvényt a nyomógombhoz, amelyben kiírjuk a beviteli mezőkben éppen olvasható szöveget a szabványos kimenetre!

Mivel a nyomógomb lenyomásához tartozó vi sszahívott függvény paraméterként megkapja a nyomógombot a memóriában kijelölő mutatót, a nyomógombbal közös ablakban található képernyőelemek könnyedén megkereshetők a `lookup_widget()` függvény segítségével. Ezt mutatja be a következő néhány sor.

```
1 #ifdef HAVE_CONFIG_H
2 # include <config.h>
3 #endif
4
5 #include <stdio.h>
6 #include <gnome.h>
7
8 #include "callbacks.h"
9 #include "interface.h"
```

```

10  #include "support.h"
11
12  void
13  on_button1_clicked(GtkButton *button,
14                    gpointer    user_data)
15  {
16      GtkWidget *entry1 = lookup_widget(GTK_WIDGET(button),
17                                       "entry1");
18      GtkWidget *entry2 = lookup_widget(GTK_WIDGET(button),
19                                       "entry2");
20      gchar *text1 = gtk_entry_get_text(GTK_ENTRY(entry1));
21      gchar *text2 = gtk_entry_get_text(GTK_ENTRY(entry2));
22
23      g_message("text1 = '%s'\n", text1);
24      g_message("text2 = '%s'\n", text2);
25  }

```

A példa egy teljes állományt mutat be, amelyet a Glade program és a programozó közösen hoztak létre. Az állomány első sorait a Glade készítette, éppen úgy, mint a függvény prototípusát. A programozónak a függvény törzsét kellett elkészítenie a 16–24. sorok megírásával.

A függvény 16–17., valamint a 18–19. sorokban lekérdezi két képernyőelem memóriabeli címét a `lookup_widget()` függvény hívásával. A függvény hívásához a programozónak a rendelkezésére állt a két képernyőelem neve (`entry1` és `entry2`), valamint az ugyanezen ablakban megtalálható nyomógomb típusú képernyőelem címe a `button` változóban. Azt is mondhatnánk, hogy a programozó azokat az `entry1` és `entry2` nevű képernyőelemeket kereste meg, amelyek éppen abban az ablakban voltak, mint amelyekben az adott nyomógomb megjelent.

A példaprogram ezek után lekérdezi a két képernyőelem aktuális szövegét a 20–21. sorokban, majd kiírja a kapott karakterláncokat a szabványos kimenetre a 23–24. sorokban.

3. fejezet

Egyszerű felhasználói felületek készítése

A következő oldalakon az egyszerűbb képernyőelemekről, a belőlük készíthető egyszerű felhasználói felületekről olvashatunk, mielőtt azonban ezeket ismertetnénk szót kell ejtenünk a GTK+ programkönyvtár néhány sajátosságáról.

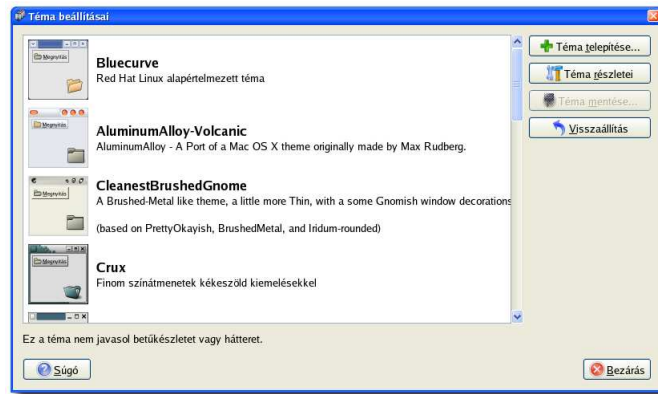
Először is tudnunk kell, hogy a GTK+ segítségével készített alkalmazások grafikus felhasználói felületének kinézete a felhasználó által menet közben módosítható, a Glade segítségével készített programjaink „témázhatók”. Ha a felhasználó elindítja a GNOME témakezelőjét a munkaasztal menüjével vagy a [gnome-theme-manager](#) parancs kiadásával, az alkalmazások kinézetét akár futtatás közben is meg tudja változtatni. A GNOME témakezelőt a 3.2. ábrán láthatjuk, a 3.1. ábrán pedig néhány példát láthatunk a használható témák közül.

Az éppen használt téma beállításainak és a használt nyelvnek megfelelően a programunk grafikus felülete nagymértékben megváltozhat, az egyes képernyőelemek magasságát, szélességét tehát nem ismerhetjük előre.

Fontos, hogy úgy szerkesszük meg a programunk grafikus felületét, hogy közben a változó körülményeket is figyelembe vegyünk. Fontos, hogy megértsük, a Glade és a GTK+ néha talán bonyolultnak tűnő viselkedésének az az oka, hogy a fejlesztőik folyamatosan figyelembe vették a változó kö-

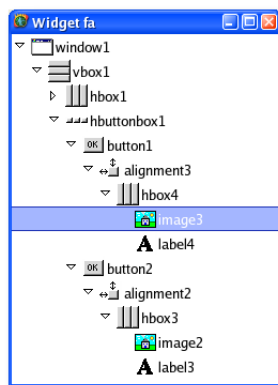


3.1. ábra. Különféle témák



3.2. ábra. A GNOME témakezelője

rülményeket, amelyek közt az általunk készített alkalmazásoknak működniük kell.



3.3. ábra. A képernyőelem fája

Tudnunk kell azt is, hogy a képernyőelemek sokféleképpen egymásba ágyazhatók, így nem mindig egyértelmű számunkra, hogy a képen látható grafikus felhasználói felületek milyen elemekből állnak. Már a kevésbé összetett felhasználói felületek készítésekor is segítségünkre lehet képernyőelemek közti viszonyokat grafikusán ábrázoló ablak, amelyek a **nézet** menü **widget fa mutatása** menüpontjával érhetünk el. Ez az ablak nagymértékben megkönnyíti az egyes képernyőelemek kiválasztását. A 3.1. ábrán látható ablak szerkezetét láthatjuk példaképpen a 3.3. ábrán, ahol jól megfigyelhetjük, hogy a nyomógombokon belül egy-egy kép és egy-egy címke található.

Szem előtt kell tartanunk azt is, hogy a GTK+ programkönyvtár objektumorientált programozási módszer-tan alapján készült, ami többek közt azt eredményezi, hogy az egyes képernyőelemek sok közös tulajdonsággal rendelkeznek.

3.1. Egyszerű képernyőelemek

A következő oldalakon olyan képernyőelemekkel foglalkozunk, amelyek kezelése egyszerű és amelyek a legegyszerűbb alkalmazásokban is elengedhetetlenek.

Azokat a képernyőelemeket, amelyek kimondottan más képernyőelemek hordozására készültek, a következő szakaszban tárgyaljuk. Szintén nem itt említjük az összetett, bonyolult kezelésű képernyőelemeket és azokat sem, amelyekre csak ritkán van szükségünk.

3.1.1. A címke

A címke minden bizonnyal a legegyszerűbb képernyőelem, hiszen csak egy egyszerű szöveget tartalmaz, amely a legtöbb alkalmazás esetén egyáltalán nem változik a program futása során. A GTK+ programkönyvtárban a címkék kezelésére használatos típus a `GtkLabel`.

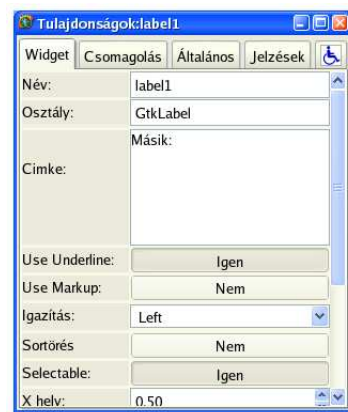
A címke példát a 3.5. ábrán láthatunk a 36. oldalon.

A 3.9. ábrán láthatjuk a Glade címkék beállítására használható ablakát. Itt legfelül a *név* és *osztály* tulajdonságok beállítására szolgáló beviteli mezőket láthatjuk. Ezek a mezők minden képernyőelemnél hasonló jelentéssel bírnak, nem csak a címkék tulajdonságainak beállításakor jelennek meg.

A *név* annak a változónak a nevét adja meg, amelybe a Glade a képernyőelem létrehozásakor a képernyőelem memóriabeli címét elhelyezi az `interface.c` állomány megfelelő függvényében. Igaz ugyan, hogy ezt az állományt nem fogjuk közvetlenül módosítani, de nyilvánvaló, hogy a C nyelv változónevekre vonatkozó korlátozásait be kell tartanunk, azaz nem használhatunk például ékezetes karaktereket vagy szóközt az itt megadott névben. A *név* mező tartalma azért is fontos lehet, mert a Glade által készített `lookup_widget()` függvény e mező alapján keres, ha tehát a későbbiekben a programunk el akarja érni ezt a képernyőelemet, akkor ezt a nevet tudnunk kell. Szerencsés tehát, ha olyan egyedi, könnyen megjegyezhető nevet választunk, ami a C programozási nyelv változóneveként is „megállja a helyét”.

A név alatt található *osztály* a képernyőelem típusát határozza meg, nem módosítható, tájékoztató jellegű.

A következő fontos beviteli mező a *címke*, ahová a címkében megjelenítendő szöveget írhatjuk be. A képernyőelemek tulajdonságainak kezelése során ez a mező is többször felbukkan, hiszen nem csak a címkében jeleníthető meg állandó szöveg, hanem például a nyomógombokban, jelölőnégyzetekben, rádiógombokban is. E mezőbe természetesen tetszőleges Unicode kódolású, akár egészen hosszú, szóközökkel és írásjelekkel tagolt szöveget is írhatunk.



3.4. ábra. A címke tulajdonságai

A címkével kapcsolatban általában felmerül a kérdés, hogy milyen nyelven készítsük el a felhasználói felületet, magyarul vagy angolul készítsük el a grafikus felhasználói felület elemeit. A többnyelvű programok készítésének összetett problémakörét a 348. oldalon található 9.4. szakaszban részletesen tárgyaljuk.

Ezek alatt az *aláhúzás* (use underline) mezőben beállíthatjuk, hogy a címkében kívánjuk-e használni az aláhúzást karakterek jelölésére. Ha igen, a címke szövegében egy aláhúzás karaktert kell elhelyeznünk az előtt a karakter előtt, amelyet alá kívánunk húzni. Az aláhúzást egyébként általában a gyorsbillentyűk jelölésére használjuk.

A *stílus* (use markup) mező meghatározza, hogy kívánjuk-e használni a betűk formázására a Pango programkönyvtár által biztosított XML-szerű nyelvet, amelynek segítségével a betűk megjelenését szabadon változtathatjuk a címként belül. Ha használjuk a stílust, a címke szövegében írhatunk olyan kifejezéseket, amelyek a betűk méretét, stílusát, színét módosítják (mint például `<span foreground="red">szöveg</span>`, vagy `<span size="larger">szöveg</span>` esetleg a dőlt betűk jelölésére használható `<i></i>` páros). A használható kifejezésekről bővebben a Pango programkönyvtár dokumentációjában olvashatunk, amelyet elérhetünk például a *devhelp* program segítségével (2.1. ábra).



3.5. ábra. Címkék, beviteli mezők és nyomógombok

Az *igazítás* mezőben beállíthatjuk, hogy a címke szövege a rendelkezésre álló terület mely részén, milyen rendezéssel jelenjen meg. A kiadványszerkesztő programok esetében megszokott rendezések közül választhatunk. A *sortó-rés* mezőben kiválaszthatjuk, hogy a címkében – ha nem áll rendelkezésre elegendő hely a szöveg számára – automatikusan új sorban kezdődjön-e a szöveg. A *kijelölhető* mezőben beállíthatjuk, hogy a címke-re kattintva megjeleníthető legyen-e a kurzor. Ha igen, a felhasználó képes lesz a címke tartalmát a vágólapra másolni, de változtatni természetesen ekkor sem tudja.

A címkéket a programunknak általában nem kell különösebb eszközökkel kezelnie, azaz léteznek ugyan a címkék megjelenésének megváltoztatására függvények, de ezekre általában nincsen szükségünk. Egyedül talán a címke szövege az, amelyet néhány alkalmazásban a futás közben meg kell változtatnunk. A címkék szövegének megváltoztatására szolgáló függvény a következő:

```
void gtk_label_set_text(GtkLabel *címke, const gchar
    *szöveg);
```

A függvény segítségével megváltoztathatjuk a címke szövegét. A függvény első paramétere a címkét, második paramétere pedig az új címkeszöveget jelöli a memóriában.

3.1.2. A kép

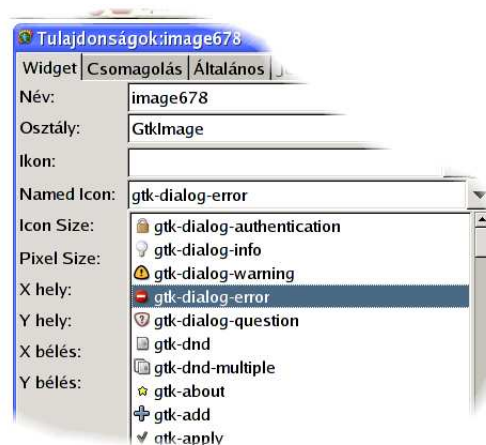
A grafikus felületen megjelenő képek a címkékhez hasonlóan passzív képernyőelemek, amelyek a számítógép vezérlésére nem használhatók, egyszerűen csak a felhasználói felületen való eligazodást segítik. A modern alkalmazások felhasználói felületein a képek sok helyen előfordulnak. Találkozunk velük a dialógusablakokban, a nyomógombokon, az eszközsávok nyomógombjain, a menükben és így tovább.

Habár a képek egyszerű képernyőelemnek tűnnek, kezelésük sokszor nem is olyan egyszerű. A GTK+ programkönyvtár többféle vektor- és pixel grafikus állományformátumot támogat, kezeli a felhasználó által beállítható témákat, képes egyszerűbb és összetettebb rajzolási műveleteket végezni a memóriába töltött képeken. Mindezek a képességei hatékony eszközzé teszik a programkönyvtárat a hozzáértő programozó kezében, ugyanakkor azonban kissé elbonyolítják a képek kezelését. Az egyszerűség kedvéért a következő néhány bekezdésben a képek kezelésére szolgáló egyszerűbb eszközöket mutatjuk csak be, az összetett eszközök használatára pedig a későbbiekben visszatérünk.

A GTK+ programkönyvtár a képek megjelenítésére a `GtkImage` típust használja, ami a `GtkWidget` osztály leszármazottjaként képernyőelemnek minősül. A `GtkImage` kezelése közben a programkönyvtár kezeli a témákat, ami azt eredményezi, hogy ha a kép témához kötött erőforrásból hozzuk létre, akkor a téma megváltoztatásakor a futó alkalmazásokban a képek megváltoznak az új téma által előírt képre, ami nyilván jelentősen módosítja az alkalmazás megjelenését.

A GTK+ programkönyvtár ugyanakkor fejlett módon kezeli a képek méretét is, az adott témának megfelelő módon többféle jellemző képméretet létrehozva. A vektor grafikus képformátumok különösen alkalmasak arra, hogy ugyanazt a képet különféle méretben jelenítsük meg a nyomógombokban, a menükben és így tovább, ha azonban a kép nem vektor grafikus formátumban áll a rendelkezésünkre, a programkönyvtár akkor is képes a több méret kezelésére. Általában egyszerűen külön-külön állományban tároljuk a különféle alapméretre előkészített képeket és mindig a megfelelő állományból állítjuk elő a képernyőn megjelenő változatot.

Amikor a Glade segítségével egy új képet helyezünk el az ablakban, a



3.6. ábra. A kép beállítása

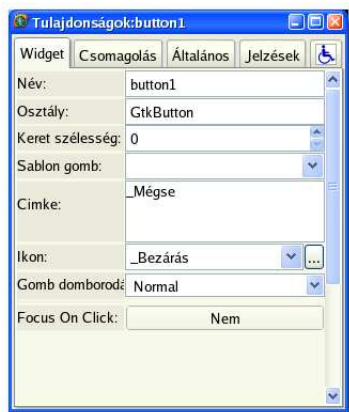
tulajdonságok beállítására szolgáló ablakban különösebb nehézség nélkül beállíthatjuk a kép megjelenését (3.6. ábra). A kép beállítását végző listák alatt hasonlóan egyszerűen állíthatjuk be a kép méretét, mégpedig néhány előre létrehozott méretkategóriából választva.

Nem ilyen egyszerű a munkánk azonban, ha a Glade által felajánlott ikonok közt egyet sem találunk, amely megfelelne az igényinknek. Ilyenkor a legegyszerűbb, ha a kép megjelenését a program futásakor, a megfelelő függvények hívásával állítjuk be. A kép beállítására használható legegyszerűbb függvények a következők.

3.1.3. A nyomógomb

A nyomógomb viszonylag egyszerű képernyőelem, aminek kevés beállítható tulajdonsága van és viszonylag kevés fajta eseményt képes kiváltani. Azonban az is igaz, hogy a nyomógombokat sok helyen, sok különféle megjelenéssel és jelentéssel használjuk, ami indokoltá teszi, hogy egy kicsit részletesebben foglalkozzunk ezzel a képernyőelemmel.

A GTK+ programkönyvtárban az egyszerű nyomógombok kezelésére használt típus a `GtkButton`, amelyre a 3.5. ábrán láthatunk példát a 36. oldalon, a Glade nyomógombok tulajdonságainak beállítására használható ablakát pedig a 3.7. ábrán találjuk.



3.7. ábra. A nyomógomb tulajdonságai

A **tulajdonságok** ablak felső részén a szokásos elemeket láthatjuk, amelyek a **név** és az **osztály** beállítására használhatók. Ezeket a tulajdonságokat minden képernyőelem esetében hasonlóan kezeljük, már ejtettünk róluk szót.

A következő mező a **keret szélessége** mező, ahol egy egész számot állíthatunk be. Érdekes módon a nyomógomb, azaz a `GtkButton` típus más képernyőelemek hordozására alkalmas, a `GtkContainer` osztály leszármazottja. Minden `GtkContainer` tulajdonságai közt beállíthatjuk a keret szélességét, ami megadja, hogy az adott képernyőelem körül – azaz a képernyőelemen kívül – hány képpontnyi üres helyet hagyjunk. Ennek a tulajdonságnak a nyomógombok esetében nem sok jelentősége van, mert ha a nyomógombokat el akarjuk távolítani a környezetüktől, akkor általában inkább az őket magukba foglaló képernyőelemek tulajdonságait kell módosítanunk.

A **sablon gomb** mezőben előre létrehozott nyomógombok közül válogathatunk. Ha itt a rendelkezésünkre álló sablonokból kiválasztunk egyet, a nyomógombon belül egy címke és egy ikon kerül egymás mellé. A nyomógomb szövege a sablonnak megfelelően, az alkalmazás számára beállított

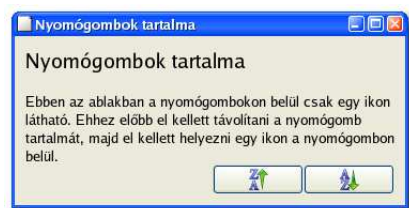
nyelven jelenik meg függetlenül attól, hogy másutt mit állítunk be. Nem csak a nyomógomb szövege változik automatikusan a nyelvnek megfelelően, hanem az ikon is a megfelelő téma szerint jelenik meg, ahogyan a 3.1. ábrán is láthatjuk.

Ha a *sablon gomb* címke mellett nem állítunk be semmit, a *címke* mezőnél állíthatjuk be a nyomógombban megjelenő szöveget. Az a karakter, amely elé aláhúzás karaktert írunk, a képernyőn aláhúzva jelenik meg. Így jelezhetjük, hogy az adott nyomógomb melyik gyorsbillentyűvel érhető el. Ha a *sablon gomb* mezőnél nem állítunk be semmit, az *ikon* mezőnél választhatunk ki képet a nyomógomb számára. Ha itt nem választunk ikont, a nyomógombban csak a címke szövege jelenik meg. Ha a címke szövegét és az ikont is beállítjuk, akkor a nyomógombban a szöveg és az ikon egymás mellett jelenik meg, mint, mint ahogyan azt a sablon gomboknál már megfigyelhettük.

A nyomógombok esetében különös fontossága van az alapértelmezett képernyőelemnek. Az alapértelmezett nyomógombot a felhasználó úgy is aktiválhatja a billentyűzettel, hogy előzőleg nem választja ki. Ha az ablak egy nyomógombjánál beállítjuk az *alapértelmezett lehet* és *alapértelmezett* tulajdonságokat a Glade tulajdonságszerkesztő ablakában, akkor a nyomógombot aktiválhatjuk az *Enter* gomb lenyomásával akkor is, ha nem az adott nyomógomb a kiválasztott képernyőelem. (Igazából azoknál a képernyőelemeknél, amelyek aktiválják az alapértelmezett nyomógombot, be kell állítanunk a *alapértelmezettet aktiválja* tulajdonságot.)

Mint már említettük, a nyomógombok más képernyőelemek hordozására alkalmas eszközök, ezért teljesen szabadon beállíthatjuk mit szeretnénk megjeleníteni a nyomógombokon belül és azt is, hogy a nyomógomb belső részének elemei egymáshoz képest hogyan helyezkedjenek el. A 3.8 például olyan nyomógombokat tartalmaz, amelyek kizárólag egy ikont tartalmaznak. Ha be szeretnénk állítani a nyomógomb tartalmát, a szerkesztőablakban a jobb egérgombbal kell kattintanunk a nyomógombra és a *nyomógombtartalom eltávolítása* (*remove button contents*) menüpontot kell választanunk. Ezek után el tudjuk helyezni a nyomógombon belül a kívánt képernyőelemet vagy a több képernyőelem elhelyezésére használható dobozt.

Ha azonban azért szeretnénk beállítani a nyomógombon belül megjelenő elemeket, hogy az alkalmazás ablakának felső részén megjelenő eszközsáv nyomógombjaiban az ikonok alatt a nyomógomb jelentésének szöveges leírása is megjelenjen, akkor felesleges ezt a bonyolult művelet-sort elvégeznünk. Az eszközsáv megjelenése, az, hogy az eszközsáv nyo-



3.8. ábra. Nyomógombok ikonnal

40

mógombjaiban a képek és a címkék hogyan jelenjenek meg, a felhasználó által beállított környezettől függ. (Az eszközsávban elhelyezett nyomógombokra a 201. oldalon, az alkalmazás-ablak bemutatásakor visszatérünk.)

A *tulajdonságok* ablak *jelzések* lapján a szokásos módon állíthatjuk be a nyomógomb jelzéseihez tartozó visszahívott függvények nevét, amelyeket a Glade a *callbacks.c* állományban el is helyez. A nyomógombok által küldött legfontosabb jelzések a következők:

clicked A nyomógomb akkor küldi ki ezt a jelzést, ha a felhasználó a egérrel vagy a billentyűzet segítségével aktiválja a nyomógombot.

A legtöbb esetben csak ezt az egy jelzést használjuk fel az alkalmazásokban.

enter A jelzés akkor keletkezik, amikor az egérmutató a nyomógomb fölé ér.

leave A jelzés akkor keletkezik, amikor az egérmutató a nyomógomb területét elhagyja.

pressed A jelzés akkor keletkezik, amikor a nyomógomb lenyomódik.

released A jelzés akkor keletkezik, amikor a nyomógomb felemelkedik. Ez a jelzés nem alkalmas a **clicked** jelzés helyettesítésére, mert akkor is jelentkezik, ha a felemelkedés nem jelez érvényes nyomógomblenyomást (mert például a felhasználó az egér gombját nem a nyomógomb felett engedte fel).

Az egyszerű nyomógombok kezelésére meglehetősen kevés függvényt használunk. Ehelyütt csak egyetlen függvényre térünk ki, ami a nyomógombban megjelenő szöveg megváltoztatására szolgál.

void gtk_button_set_label(GtkButton *gomb, const gchar *szöveg); A függvény segítségével a nyomógombban megjelenő címkét változtathatjuk meg. A függvény első paramétere a nyomógombot jelöli a memóriában, a második paramétere pedig a nyomógomb új szövegét tartalmazó karakterláncot.

10. példa. A következő sorok két visszahívott függvényt mutatnak be, amelyek közül az elsőt akkor hív a GTK+, amikor az egérmutató a nyomógomb fölé ér, a másodikat pedig akkor, amikor amikor elhagyja a területet.

```
1 void
2 on_button_window_funy_enter(GtkButton *button,
3                             gpointer user_data)
```



```

4  {
5      gtk_button_set_label(button, "OK");
6  }
7
8  void
9  on_button_window_funny_leave(GtkButton *button,
10                             gpointer    user_data)
11  {
12      gtk_button_set_label(button, "Mégsem");
13  }

```

A program roppant mulatságos, ellenállhatatlanul humoros hatását azonban csak akkor fejt ki, ha nagyobb alkalmazásba építjük és figyelmesen tanulmányozzuk a felhasználó arc kifejezését a használata közben.

3.1.4. A kapcsológomb

A kapcsológombok olyan nyomógombok, amelyek megőrzik a benyomott – vagy éppen kiengedett – állapotukat, amíg a felhasználó újra le nem nyomja őket. A kapcsológombok kezelésére a GTK+ programkönyvtár a `GtkToggleButton` típust biztosítja a programozó számára. A `GtkToggleButton` a `GtkButton` leszármazottja, így – a többalakúságnak köszönhetően – a kapcsológombokkal a nyomógombok minden művelete elvégezhető, ráadásul a kapcsológomb a nyomógomb minden tulajdonságával rendelkezik.

A Glade tulajdonságok beállítására szolgáló ablakában a kapcsológombok esetében egy lényeges tulajdonság jelenik meg a kapcsológombokhoz képest, mégpedig az *alapértelmezés szerint benyomva*, ami értelemszerűen azt határozza meg, hogy a nyomógomb induláskor lenyomott állapotban legyen-e. Kevésbé fontos, új tulajdonságként a kapcsológombot *inkonzisztens* állapotban kapcsolhatjuk. Az inkonzisztens állapotban kapcsolt nyomógomb úgy jelenik meg a képernyőn, ami a felhasználó számára érzékelteti, hogy a nyomógomb bekapcsolva és kikapcsolva sincs.

A kapcsológomb használatát megkönnyíti a következő jelzés:

toggled Ez a jelzés akkor keletkezik, amikor a felhasználó a kapcsológomb állapotát megváltoztatja, azaz a kapcsológombok ki- vagy bekapcsolja.

A kapcsológomb kezelésében a legfontosabb függvények a következők.

`void gtk_toggle_button_set_active(GtkToggleButton *kapcsológomb, gboolean bekapcsolva);` A függvény segítségével

vel a kapcsológomb állapotát beállíthatjuk, azaz a kapcsológombot be- és kikapcsolhatjuk.

A függvény első paramétere a kapcsológombot jelöli a memóriában, a második paramétere pedig meghatározza, hogy a függvény a kapcsológombot be- vagy kikapcsolja. Ha a függvénynek második paraméterként **TRUE** értéket adunk, bekapcsolja a kapcsológombot, ha **FALSE** értéket, akkor pedig kikapcsolja azt.

gboolean gtk_toggle_button_get_active(GtkToggleButton *kapcsológomb); A függvény segítségével lekérdezhetjük a kapcsológomb állapotát.

A függvény paramétere a kapcsológombot jelöli a memóriában, a visszatérési értéke pedig megadja, hogy a kapcsológomb be van-e kapcsolva. A visszatérési érték **TRUE**, ha a kapcsológomb be van kapcsolva, **FALSE**, ha nincs.

void gtk_toggle_button_set_inconsistent(GtkToggleButton *kapcsológomb, gboolean beállítás); A függvény segítségével a kapcsológombot inkonzisztens állapotba állíthatjuk. Az inkonzisztens állapotú kapcsológomb ugyanúgy működik, azaz be- és kikapcsolható, mintha nem volna inkonzisztens állapotban, de a felhasználó számára a GTK+ programkönyvtár egyértelműen jelzi, hogy a kapcsológomb sem bekapcsolt, sem pedig kikapcsolt állapotban nincs. Az inkonzisztens állapotot az alkalmazás tudja kikapcsolni és ezt általában a kapcsológomb átkapcsolásakor teszi.

A függvény első paramétere a kapcsológombot jelöli a memóriában, a második paramétere pedig meghatározza, hogy a kapcsológombot inkonzisztens állapotba akarjuk-e kapcsolni vagy éppen meg akarjuk szüntetni az inkonzisztens állapotot.

gboolean gtk_toggle_button_get_inconsistent(GtkToggleButton *kapcsológomb); A függvénnyel lekérdezhetjük, hogy a kapcsológomb inkonzisztens állapotban van-e.

A függvény paramétere a kapcsológombot jelöli a memóriában, a visszatérési értéke pedig megadja, hogy a kapcsológomb inkonzisztens állapotban van-e.

11. példa. A következő függvény bemutatja hogyan tudjuk kezelni a kapcsológomb inkonzisztens állapotát.

```

1 void
2 on_button_window_togglebutton3_toggled(
3     GtkToggleButton *togglebutton,
4     gpointer         user_data)

```

```

5  {
6      gtk_toggle_button_set_inconsistent(togglebutton,
7          FALSE);
8      if (gtk_toggle_button_get_active(togglebutton))
9          g_message("Be van kapcsolva.");
10     else
11         g_message("Ki van kapcsolva.");
12 }

```

A bemutatott függvényt a GTK+ programkönyvtár akkor hívja, ha a kapcsológombot a felhasználó átkapcsolja. A 6–7. sorban kikapcsoljuk a kapcsológomb inkonzisztens állapotát, a 8. sorban pedig lekérdezzük, hogy a kapcsológomb be van-e kapcsolva.

3.1.5. A beviteli mező

A beviteli mező rövid szövegrészletek begépelésére alkalmas viszonylag egyszerű képernyőelem. A GTK+ programkönyvtárban a beviteli mezők kezelésére használt típus a `GtkEntry`.

A beviteli mezőre példát a 3.5. ábrán láthatunk a 36 oldalon.

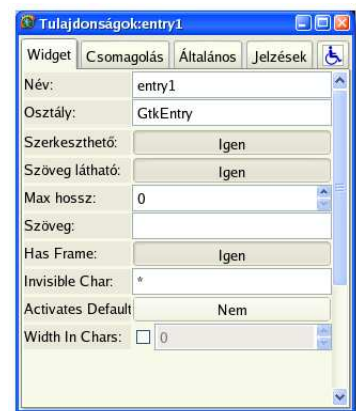
A 3.9. ábrán láthatjuk a Glade beviteli mezők beállítására használható ablakát, ahol a szokásos tulajdonságokon (*név, osztály*) kívül a legfontosabb tulajdonságok a következők.

A *szerkeszthető* mezőben beállíthatjuk, hogy a felhasználó a szövegmezőben található szöveget a billentyűzet segítségével módosíthatja-e.

A *szöveg látható* mezőben beállíthatjuk, hogy a szöveg, ami a beviteli mezőben található elolvasható legyen-e a képernyőn. Ha például a beviteli mező jelszó begépelésére szolgál, szerencsésebb, ha a *szöveg látható* tulajdonságok „nem”-re állítjuk. Ilyen esetben a *takarójel* (*invisible char*) mezőben megadott karakter jelenik meg a betűk helyén a képernyőn.

A *max hossz* mezőben azt adhatjuk meg, hogy legfeljebb hány karakternyi szöveget tartalmazhat a beviteli mező. Ennél hosszabb szöveget a felhasználó nem tud begépelni a beviteli mezőbe, mert a GTK+ erre nem ad lehetőséget. Ha itt a 0 áll, a beírt szöveg tetszőlegesen hosszú lehet.

A *szöveg* mezőben megadhatjuk, hogy mi legyen a szerkesztőmező kezdeti tartalma. Amikor a szerkesztőmező megjelenik a képernyőn ezt a szöveget már tartalmazni fogja.



3.9. ábra. A beviteli mező tulajdonságai

Az *alapértelmezettet aktiválja* (*activates default*) mezőben beállíthatjuk, hogy ez a szerkesztőmező az alapértelmezett képernyőelemet aktiválja-e, ha lenyomjuk az `[Enter]` billentyűt a szöveg begépelésekor. Ha ezt a mezőt „igen”-re állítjuk, és a felhasználó begépelte a beviteli mező tartalmát, megnyomhatja az `[Enter]` billentyűt és így aktiválhatja az ablak alapértelmezett képernyőelemét (például az OK felirattal ellátott nyomógombot.)

A beviteli mezők által küldött legfontosabb üzenetek a következők:

activate A beviteli mező akkor küldi ezt az üzenetet, ha a felhasználó lenyomta benne az `[Enter]` billentyűt. A mező ezt az üzenetet attól függetlenül elküldi, hogy egyébként aktiválja-e az ablak alapértelmezett képernyőelemét.

backspace A beviteli mező akkor küldi ezt az üzenetet, ha a felhasználó lenyomja a beviteli mezőn a `[backspace]` billentyűt.

move-cursor A beviteli mező akkor küldi ezt a jelzést, ha a felhasználó a beviteli mezőben a kurzor helyét megváltoztatja valamelyik billentyűvel.

toggle-overwrite Ez az üzenet akkor keletkezik, ha a felhasználó valamelyik irányban átkapcsol a beszúrás-felülírás üzemmódok közt.

A beviteli mezők kezelése közben a következő néhány függvény különösen hasznos lehet.

```
gchar *gtk_entry_get_text(GtkEntry *mező);
```

A függvény segítségével a beviteli mezőből kiolvashatjuk a szöveget, amelyet éppen tartalmaz. Természetesen ez a legfontosabb, beviteli mezőt kezelő függvény.

visszatérési érték A visszatérési érték egy mutató, amely azt a memóriaterületet jelöli amelyben a beviteli mezőben éppen található szöveg karakterláncként megtalálható.

Ezt a memóriaterületet semmilyen módon nem szabad módosítanunk vagy felszabadítanunk.

mező A kiolvasandó beviteli mező címe.

```
void gtk_entry_set_text(GtkEntry *mező, const gchar *szöveg);
```

A függvény segítségével megváltoztathatjuk a beviteli mezőben található szöveget.

mező A módosítandó beviteli mező címe.

szöveg A mezőben elhelyezendő új szöveg címe.

```
void gtk_editable_select_region(GtkEditable *mező, gint
    kezdet, gint vég);
```

A függvény a beviteli mezőben található karakterek kijelölésére szolgál.

mező A beviteli mezőt jelöli a memóriában. Mivel a függvény a `GtkEditable` általános típust használja, típuskényszerítésre van szükség a `GTK_ENTRY()` makró segítségével.

kezdet Az első karakter száma, amelyet ki szeretnénk jelölni. A karakterek számozása 0-tól indul.

vég Ez első karakter száma, amelyet már nem akarunk kijelölni.

3.1.6. A forgatógomb

A forgatógomb szám jellegű értékek bevitelére szolgáló, a beviteli mezőhöz nagyon hasonló képernyőelem, amelyet a 3.10. ábrán láthatunk. Ahogy megfigyelhetjük, a forgatógomb egyáltalán nem úgy néz ki, mint ahogyan a forgatógombok általában kinéznek, sokkal inkább egy beviteli mezőre hasonlítanak, ami két nyomógommbal van kiegészítve. A GTK+ programkönyvtár a forgatógombok kezelésére a `GtkSpinButton` típust biztosítja a rendelkezésünkre.

A forgatógombok tulajdonságainak beállítása közben a Glade *tulajdonságok* ablakában beállíthatjuk a forgatógombban beállítható és beírható szám legkisebb és legnagyobb értékét, a lépésközt, amivel a nyomógombok növelik és csökkentik az értéket, a lapközt, amivel a PgUp és PgDwn billentyűk növelik és csökkentik az értéket és még néhány egyszerű tulajdonságot.

A `GtkSpinButton` használatát segíti a `changed` üzenet küldése, amelyet a forgatógomb a beviteli mezőtől örökölt, s amelyet akkor küld, amikor a forgatógomb értéke megváltozik. Tudnunk kell azonban, hogy a forgatógomb nem csak akkor küldi ezt az üzenetet, ha az érték valóban megváltozott és nem is mindig azonnal érkezik az üzenet.

A `GtkSpinButton` típus a `GtkEntry` típus közvetlen leszármazottja, így a beviteli mezőknél használt függvények a forgatógombok kezelésére is használhatók, de a GTK+ programkönyvtár a forgatógombok kezelésére külön függvényeket is biztosít. A forgatógombok kezelése közben hasznosak lehetnek a következő függvények:

```
void gtk_spin_button_set_value(GtkSpinButton
    *forgatógomb, gdouble érték);
```

A függvény segítségével beállítjuk a forgatógombban megjelenő szám értékét.



3.10. ábra. A forgatógomb

A függvény első paramétere a forgatógombot jelöli a memóriában, második paramétere pedig a forgatógomb új értékét adja meg. Az, hogy az új érték hány tizedes pontossággal jelenik meg a forgatógombban, a forgatógomb beállításától függ, amelyet a Glade segítségével is megváltoztathatunk.

```
gdouble gtk_spin_button_get_value(GtkSpinButton
    *forgatógomb);
```

A függvény segítségével lekérdezhetjük a forgatógomb által képviselt értéket.

A függvény paramétere a forgatógombot jelöli a memóriában, a visszatérési értéke pedig megadja a forgatógomb értékét.

```
gint gtk_spin_button_get_value_as_int(GtkSpinButton
    *érték);
```

A függvény segítségével a forgatógomb értékét egész számként kérdezhetjük le.

A függvény paramétere a forgatógombot jelöli a memóriában, a visszatérési értéke pedig megadja a forgatógomb értékét.

3.1.7. A kombinált doboz

A kombinált doboz hasonlít a beviteli mezőhöz, ugyanakkor lehetővé teszi a felhasználónak, hogy a felajánlott szöveges értékek közül menüszerűen válasszon. Innen származik a kombinált doboz (*combobox*) kifejezés is, az összetett jellegre utal.

A GTK+ programkönyvtárban a kombinált doboz meglehetősen zavarosan jelenik meg és ezen sajnos a Glade program használata sem segít jelentősen. Mindazonáltal maga a képernyőelem a felhasználó szempontjából igen egyszerű, ezért az egyszerű képernyőelemek közt tárgyaljuk a beviteli mezővel egybeépített menüszerű eszközöket, az olvasónak pedig azt javasoljuk, hogy ha nehézségeket okoz e szakasz megértése, egyszerűen lapozzon előre és folytassa az ismerkedést a többi képernyőelemmel.

A GTK+ programkönyvtár régebbi változataiban (2.4 változat előtt) a `GtkCombo` típust biztosította a programozó számára a kombinált doboz létrehozására. Mára azonban ez a típus elavulttá vált – a dokumentáció nem javasolja új programban való használatra – és mivel a programozói felülete jelentősen eltér a modernebb eszközök felületétől, a használatát nem tárgyaljuk. Annyit érdemes azonban megjegyeznünk, hogy a `GtkCombo` típust nem szabad összekevernünk a következő oldalakon bemutatott típusokkal, mert súlyos programhibák lehetnek a típuskeveredés következményei.

Hasonlóképpen elavult eszköz a `GtkOptionMenu` típus által létrehozott egyszerű képernyőelem, ami megjelenésében nagyon hasonlít a kombinált dobozra, és amelyet szintén nem tárgyalunk részletesen.

A GTK+ programkönyvtár újabb változatai (2.4 és újabb) a `GtkComboBox` típust javasolják a kombinált doboz jellegű képernyőelem megvalósítására. A `GtkComboBox` típus leszármazottja a `GtkComboBoxEntry` típus, ami nem csak a menüszerű választást teszi lehetővé, hanem megengedi a felhasználónak, hogy a választás helyett egyszerűen beírja a mezőbe a szöveget.

A helyzetet jelentősen bonyolítja, hogy mind a `GtkComboBox`, mind pedig a `GtkComboBoxEntry` típust kétféleképpen hozhatjuk létre. Ha egyszerű programot szeretnénk írni, lehetséges, hogy szerencsésebb a kombinált doboz kizárólag szöveget kezelő változatával dolgoznunk. Ekkor változatlanul a `GtkComboBox`, illetve `GtkComboBoxEntry` típusról beszélhetünk, a képernyőelemek kezelése azonban sokkal egyszerűbb lesz. Sajnos azonban az egyszerűségnek ára is van. Ha az egyszerűsített módszerrel hozzuk létre a kombinált dobozt, később csak néhány primitív függvénnyel vezérelhetjük a képernyőelemet, előfordulhat, hogy a program fejlesztés későbbi szakaszában kell ráébrednünk, hogy a képernyőelemmel végzett munkánkat előről kell kezdenünk.

A Glade a `GtkComboBox` és `GtkComboBoxEntry` képernyőelemeket mindig az egyszerűsített – csak szöveges – módszerrel hozza létre, ezért e szakaszban ezeknek a kezelését mutatjuk be először.

A `GtkComboBox` típus kezelése közben a következő üzenetek fontosak lehetnek:

realize Minden `GtkWidget` elküldi ezt az üzenetet, amikor létrehozzuk, a `GtkComboBox` esetében azonban ennek az üzenetnek különösen fontos szerepe lehet, hiszen könnyen előfordulhat, hogy a kombinált doboz listáját a létrehozása után fel akarjuk tölteni elemekkel.

changed A `GtkComboBox` ezt az üzenetet küldi, ha a felhasználó megváltoztatja a kiválasztott elemet a listában. Ha a változást azonnal kezelni akarjuk – és nem akarunk például az ablak bezárásáig várni – ehhez az üzenethez kell csatlakoznunk.

A Glade segítségével létrehozott szerkeszthető kombinált doboz (`GtkComboBoxEntry`) és csak választásra használható kombinált doboz (`GtkComboBox`) képernyőelemek szerkesztésére az alábbi függvényeket használhatjuk:

```
void gtk_combo_box_append_text(GtkComboBox *doboz, const
gchar *szöveg);
```

A függvény segítségével a kombinált dobozhoz tartozó lista végéhez adhatunk hozzá új szövegsort.

A függvény első paramétere a kombinált dobozt, második paramétere a listához hozzáadandó szöveget jelöli a memóriában.

```
void gtk_combo_box_insert_text(GtkComboBox *doboz, gint
hely, const gchar *szöveg);
```

E függvény segítségével a kombi-

nált dobozhoz tartozó lista tetszőleges pontjára szűrhatunk be új szöveges értéket.

A függvény első paramétere a kombinált dobozt, harmadik paramétere pedig a beszűrendő szöveget jelöli a memóriában. A függvény második paramétere megadja, hogy a lista hányadik helyére kívánjuk a szöveget beszűrni. A lista elemeinek számozása 0-tól kezdődik.

`void gtk_combo_box_prepend_text(GtkComboBox *doboz, const gchar *szöveg);` Ennek a függvénynek a segítségével új szövegsort helyezhetünk el a kombinált dobozhoz tartozó lista legelső helyére, az első elem elé.

A függvény első paramétere a kombinált dobozt, második paramétere pedig a beszűrendő szöveget jelöli a memóriában.

`void gtk_combo_box_remove_text(GtkComboBox *doboz, gint hely);` A függvény segítségével a kombinált dobozhoz tartozó listából távolíthatjuk el a megadott sorszámú szövegsort.

A függvény első paramétere a kombinált dobozt jelöli a memóriában, második paramétere pedig megadja, hogy hányadik szövegsort akarjuk eltávolítani a listából. A sorok számozása 0-tól kezdődik.

`gchar *gtk_combo_box_get_active_text(GtkComboBox *doboz);` A függvény segítségével lekérdezhetjük a kombinált dobozban látható szöveget. A `GtkComboBox` esetében ez a szöveg a kombinált dobozhoz tartozó lista valamelyik eleme vagy – ha sem a program, sem a felhasználó nem választott elemet – a `NULL` értékkel jelzett üres szöveg, a `GtkComboBoxEntry` esetén azonban tetszőleges, a felhasználó által begépelte szöveg lehet.

A függvény paramétere a kombinált dobozt jelöli a memóriában, visszatérési értéke pedig egy dinamikusan foglalt memóriaterületre mutat – ahol a szöveget találhatjuk – vagy `NULL`, ha nincs megjelölt szöveg.

`void gtk_combo_box_set_active(GtkComboBox *doboz, gint hely);` A függvény segítségével meghatározhatjuk, hogy a kombinált dobozban a hozzá tartozó lista hányadik eleme jelenjen meg aktív elemként. A kombinált doboz létrehozásakor – a `realize` üzenet visszahívott függvényében – mindig érdemes ennek a függvénynek a hívásával kiválasztani valamelyik elemet, ellenkező esetben a kombinált doboz üresen jelenik meg és így a felhasználónak mindenképpen ki kell választania valamelyik elemet.

A függvény első paramétere a kombinált dobozt jelöli a memóriában, második paramétere pedig megadja, hogy a lista hányadik elemét akarjuk aktívként kijelölni. Az elemek számozása 0-tól indul.

`gint gtk_combo_box_get_active(GtkComboBox *doboz);` A függvény segítségével lekérdezhetjük, hogy a kombinált dobozban a felhasználó a hozzá tartozó lista hányadik elemét választotta ki. Ennek a függvénynek a hívása csak a `GtkComboBox` típusú – nem szerkeszthető – kombinált doboz esetében értelmes, hiszen a `GtkComboBoxEntry` típusú – szerkeszthető – kombinált doboz tartalmát a felhasználó átírhatja.

A függvény paramétere a kombinált dobozt jelöli a memóriában, a visszatérési értéke pedig megadja, hogy a hozzá tartozó lista hányadik elemét választotta ki a felhasználó. Az elemek számozása 0-tól indul.

A következő példa bemutatja hogyan tölthetjük fel a kombinált dobozt elemekkel.

12. példa. A következő függvény egy listában tárolt adatszerkezetből olvas ki szöveges értékeket és elhelyezi azokat egy kombinált dobozba, a hozzá tartozó lista szöveges értékeiként. Az ilyen jellegű függvényeket általában akkor hívjuk, amikor a kombinált doboz elkészült, de még nem jelent meg a képernyőn, azaz a hívást a *realize* kezelésére készített visszahívott függvényben kell elhelyezni.

```

1  gint
2  fill_comboboxentry_with_dictionary_titles(
3      GtkComboBoxEntry *entry)
4  {
5      GList      *li;
6      dictionary *dic;
7      gint       n = 0;
8
9      li = dictionaries;
10     while (li != NULL) {
11         dic = li->data;
12         if (dic->title != NULL) {
13             gtk_combo_box_append_text(entry, dic->title);
14             ++n;
15         }
16         li = li->next;
17     }
18
19     return n;
20 }
```

A függvény a 9. sorban a *dictionaries* nevű globális változóval jelzett

lista első elemétől indulva a 10–17. sorban található ciklus elemein halad végig.

A lista elemei – a listaelemek *data* mezői – *dictionary* típusú adatte-
rületeket jelölnek a memóriában. A *dictionary* adatszerkezet tartalmaz
egy karakteres mutató típusú, *title* nevű mezőt, amelynek értékét a 13.
sorban rendre elhelyezzük a kombinált dobozhoz tartozó listában.

3.1.8. A jelölőnégyzet

A jelölőnégyzet egyszerű két esetleg háromállapotú, önálló képernyőelem,
amely egy egyszerű eldöntendő kérdés megfogalmazására használható.
A GTK+ programnyvtárban a jelölőnégyzet kezelésére szolgáló típus a
GtkCheckButton.

A jelölőnégyzetre példát a 3.11. ábrán láthatunk.



3.11. ábra. A jelölő-
négyzet

A Glade jelölőnégyzetek tulajdonságainak beállítására
szolgáló ablakát a 3.12 ábrán láthatjuk, ahol a szokásos
(*név*, *osztály*, *keret szélesség*) tulajdonságokon kívül a
következő fontos tulajdonságokat állíthatjuk be.

A *sablon gomb* mező mellett kiválaszthatunk egy előre
elkészített szöveget és ikont. A nyomógombokhoz ha-
sonlóan a jelölőnégyzetek esetében is automatikus kö-
veti a szöveg a beállított nyelvet, valamint az ikon a be-
állított témát, ha a *sablon gomb* mezőben egy sablont
kiválasztunk.

Ha a *sablon gomb* mezőben nem választunk ki sem-
mit, a *címke* mezőben választhatjuk ki milyen szöveg
jelenjen meg a jelölőnégyzet mellett. A jelölőnégyzethez
tartozik egy címke, amely a Glade eszközeivel külön is kezelhető!

Ha a *sablon gomb* mezőben nem választottunk ki semmit, az *ikon* mező-
ben kiválaszthatunk egy ikont, amely a jelölőnégyzet mellett jelenik meg.
Ha az *ikon* mezőben egy ikont választunk, a Glade automatikusan létre-
hoz egy, a jelölőnégyzet mellett látható ikont, amely külön is kijelölhető
és amelynek tulajdonságai (például mérete) külön is állíthatók.

Az *alapért. lenyomva* mezőben beállíthatjuk, hogy a jelölőnégyzet meg-
jelenéskor be legyen-e kapcsolva, az *ismeretlen* (*inconsistent*) mezőben
pedig beállíthatjuk, hogy a jelölőnégyzet megjelenéskor a harmadik, is-
meretlen állapotot jelezze.

Az *indikátor* mezőben beállíthatjuk, hogy jelenjen-e meg a jelölőnégyzet
négyzete, amely az állapotát jelöli. Ha ezt a mezőt „nem” állapotra állítjuk,
a jelölőnégyzet állapotát a GTK+ másképpen jeleníti meg (lásd a 3.11
ábrát).

A *GtkCheckButton* típus a *GtkToggleButton* típus közvetlen leszá-
mazottja, azért a használata a 3.1.4. szakaszban bemutatott kapcsoló-
gomb használatához nagyon hasonlít. A jelölőnégyzetnek a kapcsoló-

gombhoz hasonlóan három állapota van, a **toggled** jelzést küldi az átkapcsolásakor és a legtöbb esetben ugyanazokat a függvényeket használjuk a két képernyőelem kezelésére. A könyv könnyebb megértése érdekében ebben az új környezetben is bemutatjuk a már tárgyalt eszközöket.

A jelölőnégyzetek által küldött legfontosabb jelzés a következő:

toggled A jelölőnégyzet ezt az üzenetet küldi, ha a felhasználó megváltoztatja az állapotát, azaz „átkapcsolja”.

A jelölőnégyzet használata közben a következő függvények a legfontosabbak.

```
gboolean gtk_toggle_button_get_active(GtkToggleButton
*jelölőnégyzet);
```

A függvény segítségével lekérdezhethetjük, hogy a jelölőnégyzet be van-e kapcsolva.

visszatérési érték A függvény visszatérési értéke **TRUE**, ha a jelölőnégyzet be van kapcsolva és **FALSE**, ha a jelölőnégyzet ki van kapcsolva.

jelölőnégyzet Megadja, hogy melyik jelölőnégyzetet kívánjuk lekérdezni.

A függvény nem csak a jelölőnégyzetek lekérdezésére alkalmas, a jelölőnégyzet címének átadásakor ezért esetleg használnunk kell a **GTK_TOGGLE_BUTTON()** makrót.

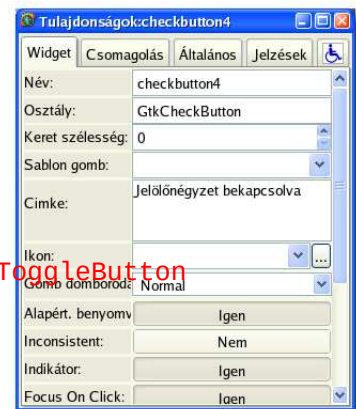
```
void gtk_toggle_button_set_active(GtkToggleButton
*jelölőnégyzet, gboolean be);
```

A függvény segítségével beállíthatjuk, hogy a jelölőnégyzet be legyen-e kapcsolva vagy sem.

jelölőnégyzet Megadja, hogy melyik jelölőnégyzetet kívánjuk átkapcsolni.

A függvény nem csak a jelölőnégyzetek lekérdezésére alkalmas, a jelölőnégyzet címének átadásakor ezért esetleg használnunk kell a **GTK_TOGGLE_BUTTON()** makrót.

be Megadja, hogy a jelölőnégyzetnek mi legyen az új állapota. Ha az átadott érték **TRUE**, a jelölőnégyzet bekapcsol, ha **FALSE** kikapcsol.



3.12. ábra. A beviteli mező tulajdonságai

3.1.9. A rádiógomb

A rádiógombok meglehetősen hasonlítanak a jelölőnégyzetekre. A különbség csak annyi, hogy amíg a rádiógombok egymástól függetlenül működnek, a rádiógombok csoportokba szervezhetők és csoportonként kölcsönösen kikapcsolják egymást. A GTK+ programkönyvtárban a rádiógombok kezelésére a `GtkRadioButton` típus használható.

A rádiógombokra példát a 3.13. ábrán láthatunk.

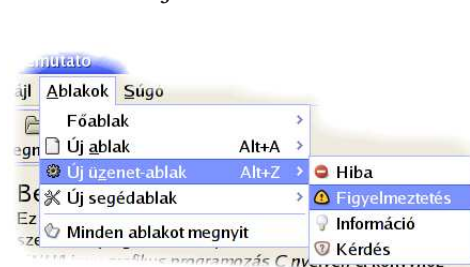


A rádiógombok kezelése majdnem mindenben megegyezik a jelölőnégyzetek kezelésével, egyetlen egy különbség van. A tulajdonságokat beállító ablakban rádiógombok esetében megjelenik egy *csort* mező, ahol beállíthatjuk, hogy a rádiógomb melyik csoportba tartozzék, azaz mely elemeket kapcsoljon ki, ha bekapcsoljuk.

3.13. ábra. A jelölőnégyzet

3.1.10. A menüsáv

A menüsáv látszólag bonyolult eszköz, a létrehozása és beállítása azonban ennek ellenére egyszerű. Ha a Glade programot használjuk az alkalmazás felhasználói felületének létrehozására, egyszerűen csak meg kell szerkesztenünk a menürendszert és a visszahívott függvények segítségével már használhatjuk is. A menüsávot mutatja be a 3.14. ábra.



3.14. ábra. Menük és almenük

A menüsávban megjelenő egyes menüpontok legfontosabb eleme a menüpont szövege, amely egyértelműen és tömören jelzi a menüpont rendeltetését. Ha a menüpontot sablon felhasználásával hoztuk létre, a GTK+ programkönyvtár gondoskodik arról, hogy a menüpont szövege a felhasználó által beállított nyelven jelenjen meg. Ha viszont nem találunk a céljainknak megfelelő sablont, a menüpont szövegét kézzel kell beírunk, a fordításról nekünk kell gondoskodnunk.

A menüpontok szövegében szokás egy betűt aláhúzni, ezzel jelezve, hogy a menüpont melyik billentyűkombinációval érhető el. Az aláhúzást úgy jelölhetjük, hogy az aláhúzandó karakter előtt egy aláhúzás jelet, a `_` karaktert helyezünk el.

Azoknak a menüpontoknak a szövege után, amelyek párbeszédablakot hoznak létre, szokás három pontot elhelyezni. A felhasználó a pontokat látva még a menüpont aktiválása előtt tudni fogja, hogy a menüpont nem azonnal feje ki hatását, lesz még alkalma beavatkozni. A három pont tehát megnyugtatja a felhasználót, ezért hasznos, érdemes használnunk.

A menüpont szövege előtt egy ikont jeleníthetünk meg a menüben. Az ikon segíti a felhasználót a menüben való eligazodásra, sokszor szükségelenné teszi, hogy a menü pontjainak szövegét végigolvassa. A GTK+ programkönyvtár gondoskodik arról is, hogy a menüpontok előtt elhelyezett ikonok mindig a felhasználó által beállított stílusnak megfelelő formában jelenjen meg. Ennek köszönhetően a felhasználó az alkalmazásokban a megszokott ikonokat látja, ami tovább könnyíti a menüben való eligazodást.

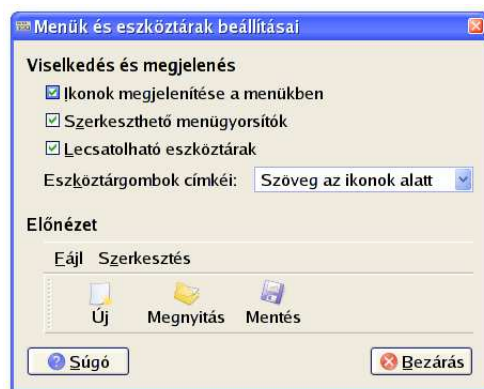
A legfontosabb menükhöz és menüpontokhoz szokás gyorsbillentyűt rendelni. Ha a menüpont szerkesztése során gyorsbillentyűt rendelünk a menüponthoz, a GTK+ programkönyvtár a gyorsbillentyű jelzését a menüpont szövege után, jobbra rendezve jeleníti meg. Ez igen hasznos szolgáltatás, hiszen így a felhasználó az alkalmazás során látja, hogy az általa használt menüponthoz milyen gyorsbillentyű tartozik és előbb-utóbb megjegyzi a leggyakrabban használt eszközökhöz tartozó billentyűkombinációit.

A menüpontok közt találunk olyat, amelyikhez almenü tartozik. Az almenü jelzésére a menüpont jobb szélén egy nyíl jelenik meg. Az almenü jelzésére szolgáló nyíl automatikusan jelenik meg, ha az adott menüpont almenüvel van ellátva. A GTK+ programkönyvtárban a menük gyakorlatilag tetszőleges mélységig egymásba ágyazhatók.

Készíthetünk olyan menüpontot is, amely saját állapottal rendelkezik, a jelölőnégyzetekhez és a rádiógombokhoz hasonlóan működik. Az ilyen menüpontok bal szélén az állapot jelzésére szolgáló ikon jelenhet meg, amelynek formája a felhasználó által beállított stílushoz igazodik.

A menük kapcsán meg kell emlétenünk a GNOME egy fontos szolgáltatását, aminek a segítségével a felhasználó egy lépésben beállíthatja az összes alkalmazás menürendszerének megjelenését. A [gnome-ui-properties](#) program (lásd a 3.15. ábrát) segítségével a menük két fontos tulajdonságát is beállíthatjuk.

Az *ikonok megjelenítése a menüben* feliratú jelölőnégyzettel a felhasználó beállíthatja, hogy az alkalmazások menüiben megjelenjenek-e az ismerős menüpontok megtalálását megkönnyítő ikonok. Különösen lassú számítógépeken hasznos, hogy az ikonokat ilyen egyszerűen kikapcsolhatjuk, hiszen ez lényegesen meggyorsíthatja a menük megjelenítését. Igaz azonban az is, hogy a GTK+ újabb változatai előbb megjelenítik a menüt és csak később



3.15. ábra. A menük beállítása

helyezik el bennük az ikonokat, ha az ikonok megjelenítése az adott számítógépen túlságosan lassú.

A **szerkeszthető menügyorsítók** jelölőnégyzetet bekapcsolva a felhasználó engedélyezheti a menüpontok mellett található gyorsbillentyűk megváltoztatását. Ehhez egyszerűen ki kell jelölnie a menüpontot – azaz le kell gördítenie a menüt, majd az egérkurzorral az adott menüpontra kell állnia – majd le kell nyomnia a gyorsbillentyűként használni kívánt billentyűkombinációt. Az új gyorsbillentyű azonnal megjelenik a menüben és természetesen azonnal használhatóvá is válik.

Talán érdemes még megemlítenünk, hogy a menük bemutatott beállításai, az ikonok elrejtése és a gyorsbillentyűk átírásának engedélyezi azonnal érvényesülnék, az éppen futó alkalmazások megjelenése és viselkedése azonnal követi a beállításokat.

A menük szerkesztése

A Glade a menük szerkesztését nagyon egyszerűvé teszi. Ha egy menüsávot helyezünk el az ablakban és kijelöljük, a **tulajdonságok** ablakban egy **menük szerkesztése...** feliratú nyomógomb jelenik meg. Ha erre a nyomógombra kattintunk megjelenik a menü szerkesztő ablak (3.16. ábra). Ebben az ablakban gyakorlatilag mindent beállíthatunk, ami a menük kezeléséhez szükséges.

Az ablak bal oldalának nagy részét elfoglaló táblázatban a menüsáv menüpontjait látjuk. A táblázatban az egyes menüpontok egymás alatt, a szerkezetet mutató hierarchikus rendben jelennek meg. A táblázat alatt található nyomógombok segítségével a kijelölt menüpontot mozgathatjuk felfelé, lefelé, illetve kijebbe és beljebb. A nyomógombok használatával bármikor átrendezhetjük a menüpontokat, ami igen megkönnyíti a program felhasználói felületének kialakítását.

Az ablak jobb oldalának felső részén, a **sablon elem** címke mellett beállíthatjuk, hogy a kijelölt menüpontot a GTK+ programkönyvtár milyen sablonból hozza létre. Ennek az eszköznek a segítségével az alkalmazásokban megszokott menüket (**fájl**, **szerkesztés** stb.) egy pillanat alatt elkészíthetjük.

A **címke** mezőben beállíthatjuk a menüpont szövegét, a **név** mezőben a menüpont kezelésekor használt változó nevét, a **kezelő** mezőben pedig a menüpont aktivizálásakor meghívott függvény nevét. Ügyeljünk arra, hogy a **név**, illetve a **kezelő** mezőben ne használjunk ékezetes karaktereket, hiszen a C programozási nyelv nem engedélyezi az ékezetes változó, illetve függvényneveket.

Az **ikon** mezőben beállíthatjuk, hogy milyen ikon jelenjen meg a menüpont előtt. Nyilvánvaló, hogy ha a menüponthoz sablont rendelünk a **sablon elem** mezőben, ez a mező nem használható, hiszen a sablon

felhasználásával létrehozott menüpontok mindig a sablon által meghatározott ikonnal jelennek meg.

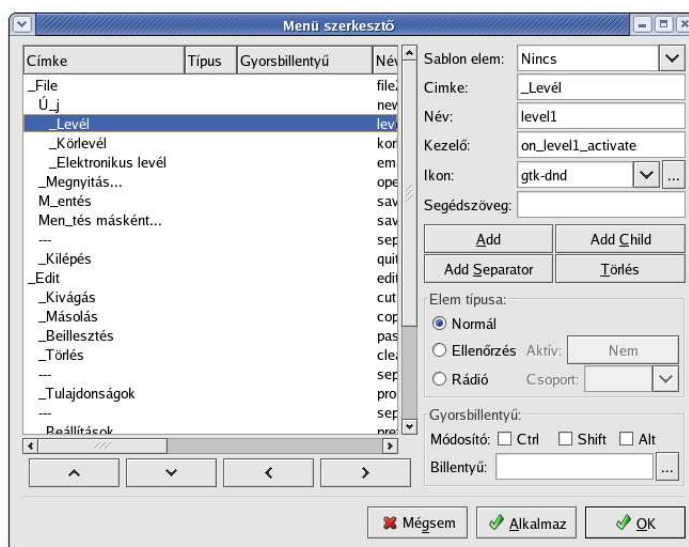
A **segédszöveg** mezőben egy rövid, néhány szavas leírást rendelhetünk a menüponthoz, amelyet a felhasználó már akkor is lát, amikor még nem aktiválta a menüpontot. A segédszöveg nagymértékben megkönnyítheti az alkalmazással ismerkedő felhasználó munkáját, ezért érdemes használni.

Az ablak jobb oldalán található **hozzáad** (add) nyomógomb segítségével új menüpontot szűrhatunk be a táblázatba kijelölt menüpont alá. Hasonló a rendeltetése a **gyermek hozzáadás** (add child) nyomógombnak, amellyel a kijelölt elem gyermekeként szűrhatunk be új menüpontot. A **gyermek hozzáadás** nyomógomb tulajdonképpen csak meggyorsítja a munkánkat, hiszen a menühöz adott menüpontot bármikor beljebb – a hierarchia mélyebb szintjére – mozgathatjuk a táblázat alatt található nyomógomb-sor segítségével.

Az ablak jobb oldalán található **elválasztó hozzáadása** (add separator) nyomógomb segítségével a menüpontokat elválasztó vonalat szűrhatunk be a bal oldalon kijelölt menüpont alá. A menüpontokat elválasztó vonal megkönnyíti a menüben való eligazodást a menüpontok csoportosításával. Elválasztójelet egyébként úgy is készíthetünk, hogy beszúrunk egy új menüpontot és egyszerűen kitöröljük a nevét.

Az ablak jobb oldalának alján található **elem típusa** részben jelölőnégyzetet és rádiógombot rendelhetünk a menüponthoz, a **gyorsbillentyű** részben pedig értelemszerűen a menüponthoz tartozó gyorsbillentyűt állíthatjuk be.

A menüszerkesztő ablak alján található **mégsem** nyomógomb segítségével bezárhatjuk a menüszerkesztőt a változtatások elvetésével. Az **OK** gombbal szintén bezárhatjuk az ablakot, de a változtatások elfogadásával.



3.16. ábra. A menü szerkesztő ablak

Az ablak alján látható *alkalmaz* nyomógomb is igen hasznos. Ezzel a gombbal érvényesíthetjük a változtatásokat anélkül, hogy az ablakot bezárnánk. Így a változtatásokat kipróbálhatjuk, megvizsgálhatjuk miképpen fog kinézni a programunk a mentés és fordítás után.

A programozó szempontjából – amint azt már említettük – a menük használata igen egyszerű. A GTK+ hívja a menüszerkesztőben megadott függvényt, amint a felhasználó aktiválta az adott menüpontot. Egyetlen nehézséget a saját állapottal rendelkező menüpontok használata jelenthet. A jelölőnégyzeteket és rádiógombokat tartalmazó menüsorok állapotát el kell kérdeznünk a visszahívott függvényben. Ennek a műveletnek az elvégzésére mutat be egy megoldást a következő példa.

13. példa. A jelölőnégyzettel ellátott menüsorok állapota kideríthető, ha a menüpontra *GtkCheckMenuItem* típussal hivatkozunk. Ezt mutatja be a következő programrészlet, amely egy jelölőnégyzettel ellátott menüsor visszahívott függvénye.

```

1 void
2 on_cimzett_activate(GtkMenuItem *menuitem,
3                     gpointer      user_data)
4 {
5     GtkCheckMenuItem *item;
6     item = GTK_CHECK_MENU_ITEM(menuitem);
7     if (gtk_check_menu_item_get_active(item))
8         printf("%s(): Menü bekapcsolva.\n", __func__);
9     else
10        printf("%s(): Menü kikapcsolva.\n", __func__);
11 }

```

Ugyanez a módszer változtatás nélkül használható akkor is, ha nem jelölőnégyzetet, hanem rádiógombot tartalmazó menüsort kezelünk a visszahívott függvénnyel.

3.1.11. A felbukkanó menü

A felbukkanó menü (*popup menu*) olyan menü, amely az egérgomb megnyomásának hatására az egérmutató aktuális helyén jelenik meg. A felbukkanó menü általában kontextusérzékeny, azaz hatása arra a képernyőelemre vonatkozik, amelyikre kattintva a menüt megjelenítettük. A 3.17. ábrán például a felbukkanó menü hatása arra az ikonra vonatkozik, amelyre kattintottunk, amelyen a felbukkanó menü bal felső sarka található.

Ha a programunkat a Glade segítségével készítjük, a felbukkanó menüket is ugyanúgy szerkeszthetjük, mint a menüsorban található menüket. Egyszerűen válasszuk ki a *felbukkanó menü* ikont a *paletta* ablakból, vagy válasszuk ki a már létrehozott felbukkanó menüt dupla kattintással a Glade főablakában és máris szerkeszthetjük a felbukkanó menü menüpontjait.

A Glade minden felbukkanó menü számára létrehoz egy függvényt, amelyet hívva a felbukkanó menüt létrehozhatjuk. A menüt létrehozó függvény nevét megkaphatjuk, ha a felbukkanó menü neve elé elhelyezzük a *create_* (*create*, létrehoz) előtagot.

A felbukkanó menük használata során a legfontosabb függvények a következők:

GtkWidget *create_xxx(void); A Glade által létrehozott függvény, amely a felbukkanó menü létrehozását végzi. A függvény nevében az *xxx* helyén a felbukkanó menü neve található. Fontos lehet tudnunk, hogy a függvény létrehozza ugyan a felbukkanó menüt, de azt a képernyőn nem jeleníti meg. A menü megjelenítését külön függvény hívásával kell kérnünk.

A függvény visszatérési értéke a létrehozott menüt jelöli a memóriában.

void gtk_menu_popup(GtkMenu *menü, GtkWidget *szülő_menü, GtkWidget *szülő_menüsor, GtkMenuPositionFunc függvény, gpointer adat, guint gomb, guint32 időpont);

A függvény segítségével az előzőleg létrehozott menüt felbukkanó menüként megjeleníthetjük a képernyőn. A függvény a menü megjelenítése után azonnal visszatér, a program futása folytatódik. Ha a felhasználó kiválasztja a felbukkanó menü valamelyik menüpontját, a Glade menüszerkesztőjében beállított visszahívott függvény lefut.

A függvény paraméterei a következők:

menü A menüt – amelyet meg akarunk jeleníteni – jelöli a memóriában.

szülő_menü A menühöz tartozó szülőmenü, ha a menü egy másik menü almenüjeként jelenik meg. Ezt a paramétert általában nem kell használnunk, a helyén a **NULL** értéket adhatjuk meg.



3.17. ábra. A felbukkanó menü

szülő_menüsor A szülő menü adott menüsorát jelöli a memóriában. A legtöbb esetben itt is **NULL** értéket adhatunk meg.

függvény A menü megjelenési helyét kiszámító függvény mutatója. A legtöbb esetben itt is **NULL** értéket adhatunk meg.

adat A menü megjelenési helyét kiszámító függvénynek átadandó adatokat jelölő mutató. A legtöbb esetben itt a **NULL** értéket használjuk.

gomb Annak az egérgombnak a sorszáma, amelynek hatására a menü megjelenítését kérjük. Ha a menü nem az egér nyomógombjának lenyomása hatására jelenik meg, használhatjuk a 0 értéket.

időpont A menü megjelenését kiváltó esemény bekövetkeztének időpontja a GTK+ saját időábrázolási rendszere szerint. Ha az esemény bekövetkezésének ideje ismeretlen, használhatjuk a `gtk_get_current_event_time()` függvény által visszaadott értéket.

A felbukkanó menü létrehozását általában olyan visszahívott függvényben végezzük el, amelyet a GTK+ valamilyen képernyőelemre való kattintáskor hív. A következő üzenet például nagyon alkalmas felbukkanó menü létrehozására:

button_press_event Ezt az üzenetet gyakorlatilag bármilyen típusú képernyőelem képes kiváltani. Az üzenet az egér valamilyik nyomógombjának lenyomásakor keletkezik, ha az egérmutató az adott képernyőelem felett helyezkedik el¹.

Az üzenet hatására visszahívott függvény visszatérési értéke logikai típusú (**gboolean**), fontos szerepet kap az adott képernyőelem életében.

Ha az általunk készített visszahívott függvény visszatérési értéke logikai igaz (**TRUE**), a GTK+ az egér lenyomását nem kezeli. Az igaz visszatérési értékkel tehát azt jelezhetjük, hogy az egér gombjának lenyomását „elintéztük”, azzal a programkönyvtárnak nem kell foglalkoznia. Ha a visszatérési érték hamis (**FALSE**), a GTK+ programkönyvtár az egérgomb lenyomását az adott képernyőelemnek megfelelő módon feldolgozza, azaz az egérgomb lenyomása kifejti szokásos hatását.

Az egér lenyomásakor hívott függvény paraméterként megkapja a lenyomás körülményeit jelző **GdkEventButton** adatszerkezetet. Ennek a típusnak a legfontosabb elemei a következők:

¹A GTK+ elnevezési rendszere szerint a *button* (gomb) mindig az mutatóeszköz, az egér nyomógombját jelenti. A billentyűzet gombjait a *key* (billentyű) szóval jelöli a program

GdkEventType type Az esemény típusa, amely egér gombjának lenyomása esetén mindig **GDK_BUTTON_PRESS**.

guint32 time Az esemény bekövetkeztének időpontja a GTK+ saját időformátumában kifejezve.

gdouble x, y Az esemény bekövetkeztének pontos helye a képernyőelem saját koordinátarendszerében. A koordinátarendszer 0,0 helye a képernyőelem bal felső sarkában van, az *x* és *y* értékek jobbra, illetve lefelé növekednek képpontokként 1-el.

gdouble x_root, y_root Az esemény bekövetkeztének helye a képernyő saját koordinátarendszerében, amely hasonlóan épül fel, mint a képernyőelem koordinátarendszere, de azzal ellentétben a 0,0 pont a képernyő bal felső sarkában található.

guint button Az eseményt kiváltó egérgomb sorszáma. Az 1-es sorszámot viseli az egér bal oldali, a 2-es sorszámot a középső, illetve a 3-as sorszámot a jobb oldali nyomógomb.

A következő példa bemutatja hogyan tudunk felbukkanó menüt létrehozni és megjeleníteni a képernyőn. A példa nem tartalmazza azokat a visszahívott függvényeket, amelyek a felbukkanó menü egyes menüpontjainak kiválasztásakor futnak, mivel ezek a függvények a szokásos módon működnek.

14. példa. A következő függvény a Glade programmal készült, egy tetszőleges típusú képernyőelemhez tartozik, a **button_press_event** esemény bekövetkeztekor fut le. A bemutatott függvény egy felbukkanó menüt hoz létre és jelenít meg az egér jobb gombjának lenyomása után.

```

1  gboolean
2  on_iconview_button_press_event(GtkWidget      *widget,
3                                GdkEventButton *event,
4                                gpointer        user_data)
5  {
6      GtkWidget *menu_widget;
7
8      /*
9       * Ha az esemény nem egérgomb lenyomás vagy nem a
10      * jobb egérgomb lenyomása, visszatérünk és kérjük,
11      * hogy az eseményt a GTK+ kezelje.
12      */
13      if (event->type != GDK_BUTTON_PRESS ||
14          event->button != 3)
15          return FALSE;

```

```

16
17      /*
18       * Létrehozzuk a menüt és megjeleníttetjük ott, ahol
19       * a felhasználó lenyomta az egérgombot.
20       */
21      menu_widget = create_popup_menu1();
22      gtk_menu_popup(GTK_MENU(menu_widget),
23                    NULL,
24                    NULL,
25                    NULL,
26                    NULL,
27                    event->button,
28                    event->time);
29      /*
30       * Mindent megcsináltunk, nem maradt tennivaló.
31       */
32      return TRUE;
33  }
```

Figyeljük meg, hogy a `gtk_menu_popup()` függvény paramétereit milyen egyszerűen adhatjuk meg! Csak néhány paraméterre van szükség és azokat is kinyerhetjük a `GtkEventButton` típusú eseményleíróból.

3.1.12. Az eszközsáv

Az eszközsáv (*toolbar*) az ablak felső részén megjelenő vízszintes sáv, amely általában ikonokkal ellátott nyomógombokat tartalmaz az alkalmazás legfontosabb szolgáltatásainak elérésére. Az eszközsávra mutat be példát az 52. oldalon látható 3.14. ábra. Amint látjuk az eszközsávban nem csak nyomógombokat helyezhetünk el, itt is a szokásos módon használhatjuk a képernyőelemeket.

Az eszközsáv létrehozására és kezelésére a 201. oldalon, az alkalmazásablak bemutatásakor részletesen is kitérünk.

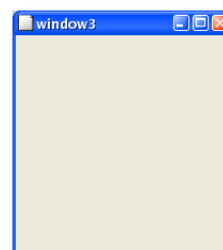
3.2. Doboz jellegű elemek

A következő oldalakon olyan képernyőelemekről lesz szó, amelyek kimondottan más képernyőelemek hordozására készültek. Amint azt már láttuk, a nyomógombban is elhelyezhetők más képernyőelemek, de a nyomógomb elsődleges feladata nem képernyőelemek hordozása, míg az itt bemutatott eszközök elsődlegesen ilyen feladatokat látnak el.

3.2.1. Az ablak

A legfontosabb doboz jellegű képernyőelem vitathatatlanul az ablak, amely lehetővé teszi, hogy az alkalmazás a felhasználó által kényelmesen kezelhető képernyőelemeket helyezzen el a képernyőn. A GTK+ programkönyvtár az ablakok kezelésére a `GtkWindow` típust használja.

A 3.18. ábrán egy üres ablakot láthatunk. Az ablak középső részén található üres fehér terület (*client area*) az alkalmazás fennhatósága alá tartozik, kizárólag az ablakot megnyitó alkalmazás számára van fenntartva. Az alkalmazás számára fenntartott területet egy szegély veszi körül. Ennek a szegélynek a kezelését az ablakkezelő (*window manager*) végzi. Az alkalmazás korlátozottan beleszólhat ugyan, hogy az ablakkezelő miképpen végezze feladatát, de a szegély létrehozása és kezelése nem az ő feladata. Éppen ezért az ablak körül látható szegély kinézete, használata független az alkalmazástól.



3.18. ábra. Az ablak

Ebből következik, hogy a Glade által készített alkalmazások kissé másképp jelennek meg az egyes ablakkezelőkkel futtatva őket, viszont futnak a különféle ablakkezelő rendszereken (például KDE, FVWM, Afterstep stb.). Az alkalmazásnak a GTK+ (és még néhány más) programkönyvtárra van szükségük, az azonban, hogy melyik ablakkezelőt használjuk, mellékes.

Az ablak tulajdonságainak beállítására szolgáló ablak a 3.19. ábrán látható. Itt a szokásos tulajdonságokon (*név*, *osztály*) kívül a következő fontos jellemzőket állíthatjuk be.

A *keret szélesség* mező sok más képernyőelem esetében megjelenik, de az ablakok esetében különösen nagy jelentősége van. A *keret szélesség* mezőben megadhatjuk, hogy az ablak belső szélén hány képpontnyi helyet hagyjuk az ablak tartalma körül. Ez a jellemző tehát a nyomtatott szöveg kezelésekor használt margóhoz hasonlít. Érdeemes mindig legalább 10 képpontnyi keretet beállítani az ablakokhoz, mert különben egyszerűen csúnya lesz az alkalmazás megjelenése.

A *fejléc* mezőben beállíthatjuk, hogy az ablak felső részén – a címsorban – milyen szöveg jelenjen meg. A címsor megjelenítése az ablakkezelő feladata, ezért soha nem lehetünk biztosak abban, hogy ez a szöveg meg is jelenik a képernyőn, és abban sem, hogy ha megjelenik, akkor olvasható lesz. Lehetséges például, hogy a felhasználó által használt ablakkezelő nem képes megjeleníteni az ékezetes karaktereket. Érdeemes megemlítenünk azt is, hogy az alkalmazás a saját ablakai címsorát bármikor átállíthatja a `gtk_window_set_title()` függvény segítségével.

Fontos információt tartalmaz a *típus* mező is. Itt állíthatjuk be azt, hogy az ablakot az ablakkezelő kezelje-e, vagy egyszerűen hagyja figyelmen kívül. Ha itt a *top level* értéket állítjuk be, az ablakot az ablakkezelő el-

látja kerettel, a felhasználó pedig keret segítségével kezelheti. Ha a **típus** mezőben a **popup** értéket állítjuk be, az ablakkezelő egyszerűen figyelmen kívül hagyja az ablakot, amely ennek megfelelően nem kap szegélyt, amelynek a segítségével mozgatni, méretezni lehetne. Nyilvánvaló, hogy az alkalmazások ablakai számára a **top level** típust kell használnunk, míg a **popup** stílus azok számára a képernyőelemek számára van fenntartva, amelyek valamelyik ablakhoz vannak ugyan rendelve, de elhagyhatják a befogadó ablak területét (például menük).



3.19. ábra. Az ablak beállítási

A **pozíció** mezőben beállíthatjuk, hogy az ablak hol jelenjen meg. Sokszor kényelmes, ha az új ablak az egérkurzornál jelenik meg, mert így könnyű azonnal bezárni, vagy esztétikus, ha a képernyő közepén tűnik fel. Mindenesetre az itt beállított érték csak egy kérést fogalmaz meg az ablakkezelőnek, hogy valójában hol jelenik meg az ablak, az az ablakkezelő beállításaitól is függhet.

A **modális** (*modal*) mezőben beállíthatjuk, hogy az ablak mennyire akadályozza a felhasználót a más ablakokra való átkapcsolásban. A modális ablakok amíg a képernyőn vannak akadályozzák, hogy a felhasználó az adott alkalmazás más ablakait használja, azokat érzéketlenné teszik. (Szerencsére a GNU/Linux rendszereken ismeretlen az úgynevezett „rendszer modális” ablak fogalma. Ez az ablaktípus képes a mögötte látható teljes képernyőt letiltani, hogy a fogát erőteljesen csikorgató felhasználó kénytelen legyen az adott ablakkal foglalkozni.)

Az **alap szélesség** és **alap magasság** mezőkben beállíthatjuk, hogy mennyi legyen az ablak képpontban mért szélessége és magassága kezdetben. Ha ezeket a jellemzőket nem állítjuk be, az ablak éppen akkora lesz, amekkora területen a benne ta-

lálható elemek elférnek. A legtöbb esetben felesleges beállítani a magasságot és a szélességet, hiszen nem tudhatjuk, hogy az adott felhasználói beállítások mellett (például különlegesen nagy betűk) mekkora helyre van szükség az adott ablak megjelenítéséhez.

A **méretezhető** (*resizable*) mezőben beállíthatjuk, hogy a felhasználónak lehetővé kívánjuk-e tenni, hogy az ablak méretét az alkalmazás futása közben megváltoztassa.

Az **automata megsemmisítés** (*auto-destroy*) a látszat ellenére nem túl bonyolult vagy veszélyes tulajdonságot takar, egyszerűen azt jelenti, hogy az ilyen tulajdonsággal rendelkező ablakok megsemmisülnek, ha az őket létrehozó ablak (ugyanannak az alkalmazásnak egy másik ablaka) meg-

semmisül.

Az ablakok kezelésére a 179. oldalon található 7.1. szakaszban, a több ablakot használó alkalmazások tárgyalása során visszatérünk.

3.2.2. A függőleges doboz

Amint láttuk a képernyőelemek elhelyezésére alkalmas képernyőelemek legtöbbje (például ablak, nyomógomb) csak egyetlen képernyőelem hordozására képes. Az ilyen képernyőelemekben is elhelyezhetünk azonban több képernyőelemet, ha függőleges vagy vízszintes dobozt használunk. A függőleges és vízszintes dobozok szerepe éppen az, hogy egymás alatt, illetve egymás mellett több képernyőelemet hordozzanak. Mivel a függőleges és vízszintes dobozokban is lehetnek függőleges, illetve vízszintes dobozok, tetszőlegesen összetett szerkezetek építhetők a segítségükkel.

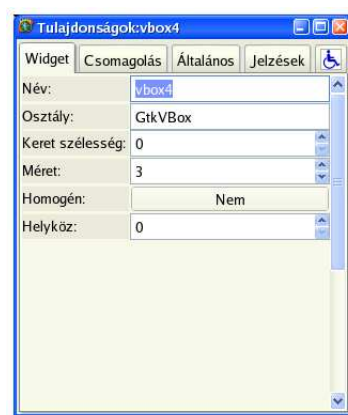
A függőleges doboz segítségével képernyőelemeket helyezhetünk egymás alá. A GTK+ programkönyvtár esetében a függőleges dobozok kezelésére használható típus a `GtkVBox`. A függőleges dobozok tulajdonságainak beállítására használható ablakot a 3.20. ábrán láthatjuk. Itt a szokásos tulajdonságok mellett (*név, osztály*) a következő fontos tulajdonságokat is beállíthatjuk.

A *keret szélessége* más képernyőelemek esetében is állítható, a függőleges dobozok esetében azonban különösen fontos ez a jellemző. Ahogy az ablakoknál már megfigyelhettük, ez a jellemző megadja, hogy a doboz belső részén hány képpontnyi helyet hagyjunk üresen. Itt általában érdemes mintegy 5 képpontnyi távolságot beállítani. Fontos megjegyeznünk, hogy a függőleges doboz szegély csak egy keret, nagyobbra állítása nem növeli a távolságot az egymás alatt elhelyezett elemek között.

A *méret* jellemző megadja, hogy hány elem elhelyezésére alkalmas a doboz, azaz a doboz területe hány egymás alatti részterületre oszlik.

A *homogén* jellemző megadja, hogy az egyes területek kötelezően azonos méretűek-e. Esztétikai okokból sokszor azonos méretű helyre helyezzük a különféle méretű elemeket, hogy így táblázatos formában egymás alá vagy mellé kerüljenek az elemek. Ilyen esetben ezt a mezőt „igen” értékre kell állítanunk.

A *helyköz* jellemző megadja, hogy az egyes részek közt hány képpontnyi helyet hagyunk ki. Általában ezt a jellemzőt is érdemes néhány képpontnyi értékkel növelnünk, hogy az egyes képernyőelemek ne kerüljenek túl közel egymáshoz.



3.20. ábra. A függőleges doboz tulajdonságai

3.2.3. A vízszintes doboz

A vízszintes doboz képernyőelemek egymás mellé helyezését teszi lehetővé. A GTK+ könyvtár esetében a vízszintes dobozok használatára a `GtkHBox` típus szolgál.

A vízszintes dobozok értelemszerűen mindenben megegyeznek a függőleges dobozokkal, az egyetlen különbség a bennük található elemek vízszintes elrendezése.

3.2.4. A táblázat

A táblázat hasonlít a függőleges és a vízszintes dobozokhoz, de nem csak sorai, illetve oszlopai vannak, hanem sorai és oszlopai is. A GTK+ táblázatok kezelésére használható típusa a `GtkTable`.

A táblázat kezelésére használható függvények legtöbbjét csak akkor találhatjuk hasznosnak, ha a felhasználói felületet nem a Glade segítségével, hanem „kézzel” a megfelelő függvények hívásával hozzuk létre.

3.2.5. A keret



3.21. ábra. A keret

A keret igen egyszerű, egyetlen képernyőelem hordozására alkalmas képernyőelem, amely segítségünkre lehet az ablak tartalmának csoportosításában. A GTK+ keretek kezelésére használható típusa a `GtkFrame`.

A keret legfontosabb feladata, hogy néhány képernyőelemet egybefogva, a többi képernyőelemtől elkülönítve jelenítsen meg. A keret rendelkezik egy címkével, amely általában a képernyőelemek csoportját látja el külön névvel, hogy könnyebben hivatkozhas-

sunk rá. A keret egy lehetséges megjelenési formáját mutatja be a 3.21. ábra.

A keretek használata közben hasznos lehet a következő függvény:

```
void gtk_frame_set_label(GtkFrame *keret, const gchar
    *szöveg);
```

A függvény segítségével beállíthatjuk a keret címkéjeként megjelenő szöveget. A függvény második paramétere egy szöveget jelöl a memóriában, ami az első paraméterként jelzett keret címkéjeként a képernyőn megjelenik.

Ha a címke szövegét félkövér betűtípussal szeretnénk megjeleníteni, akkor használhatjuk a `<b>szöveg</b>` formát ennek jelzésére.

3.3. Általános eszközök

A GTK+ programkönyvtár használata során sokszor hívunk olyan függvényeket, amelyek többféle adatszerkezetet, többféle képernyőelemet is képesek kezelni. A következő oldalakon ezek közül a függvények közül vesszük sorra a legfontosabbakat.

3.3.1. Az objektum

A GTK+ minden képernyőeleme – a `GtkWidget` típus – a G programkönyvtár objektumtípusára, a `GObject` típusra épül. A G programkönyvtár a `GObject` típust biztosítja az objektumorientált programozás alaptípusaként.

A G programkönyvtár a `GObject` típusú változók – az objektumok – számára egy hivatkozásszámlálót rendel, hogy az objektumok létrehozását és megsemmisítését támogassa. A programkönyvtár minden objektumhoz egy hivatkozásszámlálót rendel, ami kezdetben 1 és amelyet a programunk növelhet, illetve csökkenthet. Ha a hivatkozásszámláló értéke 0, a programkönyvtár az adott objektumot megsemmisíti, a hozzá tartozó memóriaterületet felszabadítja. Ezzel az eszközzel az objektumok megsemmisítése automatizálható, a programozónak tehát nem kell nyilvántartania azt, hogy az egyes objektumokhoz tartozó memóriaterületeket mikor lehet felszabadítani.

A G programkönyvtár hivatkozásszámlálójáról tudnunk kell, hogy annak közvetlen kezelése a GTK+ képernyőelemek esetében nem szerencsés. A GTK+ a képernyőelemek hivatkozásszámlálójának kezelésére saját rendszert valósít meg, amelyet a G programkönyvtárral való közvetlen kezelés megzavar. Mindazonáltal léteznek olyan típusok is, amelyeket a G programkönyvtár hivatkozásszámlálók kezelésére alkalmas függvényeivel kezelhetünk.

A G programkönyvtár a hivatkozásszámlálókon kívül egy adatterületet is rendel minden objektumhoz. Az adatterülethez a programunk névvel és értékkel rendelkező változó szerű bejegyzéseket adhat hozzá és ezeket a változókhoz hasonló bejegyzéseket bármikor lekérdezhet. Ez a rendszer a GTK+ programkönyvtár képernyőelemeinek kezelése során is igen hasznos lehet, hiszen a segítségével könnyen elkerülhetjük a globális változók használatát.

Az általános hatókörű, globális átlózők használata igen szerencsétlen a grafikus felhasználói felületek készítése során, hiszen a globális változó használata lehetetlenné teszi, hogy a globális változó segítségével leírt képernyőelemből – például ablakból – egy időben többet is megjelenítsünk a képernyőn. Ha például egy globális változóban tároljuk a szövegszerkesztőben megnyitott állomány nevét, akkor – mivel csak egy

globális változónk van – csak egy ablakot jeleníthetünk meg.

Megoldást jelenthet, ha a szövegszerkesztő képernyőelem, vagy a szövegszerkesztőt magába foglaló ablak adatterületéhez a G programkönyvtár segítségével egy bejegyzést rendelünk, ami tartalmazza az állománynevet. Így minden szövegszerkesztőhöz, minden ablakhoz saját állománynevet rendelhetünk és mindig egyértelmű lesz, hogy melyik állománynev tartozik az adott szövegszerkesztőhöz.

A **GObject** típus kezelésére használható függvények közül a legfontosabbak a következők.

gpointer g_object_get_data(GObject *objektum, const gchar *név); A függvény segítségével az objektumhoz tartozó adatterület bejegyzései közt kereshetünk név szerint.

A függvény első paramétere az objektumot jelöli a memóriában, a második paramétere pedig a keresett bejegyzés nevét adja meg.

A függvény visszatérési értéke az objektumhoz tartozó adatterület adott nevű bejegyzését jelöli a memóriában. Ha a paraméterként átadott néven nem található bejegyzés, a visszatérési érték **NULL**.

void g_object_set_data(GObject *objektum, const gchar *név, gpointer adat); A függvény segítségével az objektumhoz tartozó adatterülethez adhatunk új bejegyzést vagy egy már meglévő bejegyzés értékét változtathatjuk meg.

A függvény első paramétere az objektumot jelöli a memóriában, a második paramétere pedig a létrehozandó vagy megváltoztatandó bejegyzés nevét határozza meg. Ha a második paraméter által meghatározott néven még nem létezik bejegyzés, a G programkönyvtár létrehozza, ha pedig már létezik ilyen bejegyzés, akkor egyszerűen megváltoztatja az értékét.

A függvény harmadik paramétere a bejegyzés értékét jelöli a memóriába. Az objektumot adatterületeinek minden bejegyzése egy mutatóértéket hordoz, ami tetszőleges típusú érték hordozására teszi képessé az adatterületet.

void g_object_set_data_full(GObject *objektum, const gchar *név, gpointer adat, GDestroyNotify függvény); A függvény a **g_object_set_data()** függvényhez hasonlóan új bejegyzést hoz létre vagy megváltoztatja a már meglévő bejegyzés értékét, az egyetlen különbség a negyedik paraméterként átadott mutató kezelésében mutatkozik.

A negyedik paraméter egy függvényt jelöl a memóriában, amelyet a G programkönyvtár akkor hív, ha az adatterület bejegyzése megváltozik, azaz, amikor a bejegyzést megváltoztatjuk vagy az adatterület

objektumát megsemmisítjük. A negyedik paraméter által jelölt függvényt nyilván arra használhatjuk, hogy az adatterület bejegyzése által kijelölt adatterületet felszabadítsuk vagy a hivatkozásszámlálóját csökkentjük.

A grafikus felülettel rendelkező programokban igen hasznosnak bizonyulhatnak a `g_object_set_data()` és a `g_object_get_data()` függvények, mégpedig azért, mert a visszahívott függvények paraméterlistája kötött, a globális változók használata viszont igen hátrányos következményekkel jár.

Amikor egy új ablakot, új felbukkanó menüt jelenítünk meg, akkor általában olyan információkkal rendelkezünk amelyekre szükségünk lehet az ablak, a menü kezelése során. Amikor például a felhasználó számára egy felbukkanó menüt jelenítünk meg, mert a programunk egy ikonjára kattintott, nyilvánvalóan tudjuk, hogy a felbukkanó menü melyik állományt jelző ikon felett jelent meg. A felbukkanó menü egyes menüpontjainak visszahívott függvényeiben szükségünk van erre az információra, tudnunk kell, hogy a felhasználó melyik állományt akarja törölni, átnevezni, megnyitni és így tovább.

Kezdő programozónak komoly nehézséget jelent a kiegészítő információk átadása a visszahívott függvénynek. A visszahívott függvények hívását a Glade által készített kód alapján a GTK programkönyvtár végzi, így a visszahívott függvény paramétereinek megváltoztatása nehézkes (bár erre volna mód), különösen, ha a visszahívott függvényt a Glade segítségével rendeljük a jelzésekhez. Használhatnánk globális változókat is, amelyekben az ablakok, menük megjelenítésekor tárolhatnánk a kiegészítő információkat és amelyekből a visszahívott függvények kiolvashatnák azokat. A globális változók használata néha megoldás jelenthet, de egyrészt nem túl kellemes a globális változóktól hemzsegó program fejlesztése, másrészt igen nehéz a globális változókból tárolt kiegészítő információkat kezelni, ha megengedjük, hogy az ablakokból egyszerre több is létezzen (bár természetesen ez is megoldható).

Igen egyszerű, ugyanakkor problémamentes megoldást a `g_object_set_data()` és a `g_object_get_data()` függvények jelenthetnek, ahogyan a következő példa is bemutatja.

15. példa. *Tegyük fel, hogy a programunk tartalmaz egy ablakot hálózati erőforrások szerkesztésére és ez az ablak másképpen kell, hogy viselkedjen akkor, amikor egy meglévő erőforrást szerkesztünk és megint másképpen, amikor egy új erőforrást hozunk létre.*

A következő függvény az új hálózati erőforrás létrehozására használt menüpont visszahívott függvényét mutatja be.

```
1 void
2 on_new_network_resource_activate(GtkMenuItem *menuitem,
```

68

```

3                                     gpointer    user_data)
4 {
5     GtkWidget *window;
6     window = create_network_resource_window();
7
8     g_object_set_data(G_OBJECT(window),
9                       "purpose",
10                      "insert");
11     gtk_widget_show(window);
12 }
```

A függvény a 6. sorban létrehozza az új ablakot, amely egy *GtkWidget* típusú objektum. Mivel a *GtkWidget* típus a *GObject* típus leszármazottja, adatot kapcsolhatunk hozzá a *g_object_set_data()* függvény segítségével, amelyet meg is teszünk a 8–10. sorokban.

Mivel új hálózati erőforrásról van szó, nem pedig létező erőforrás szerkesztéséről, szükségtelen, hogy az ablakhoz hozzárendeljünk egy meglévő erőforrást leíró adatszerkezetet. Egyszerűen csak azt a tényt kell tárolnunk, hogy az ablakot új hálózati erőforrás létrehozására hoztuk létre.

Nyilvánvaló, hogy az ilyen módon elkészített ablak és az abban található összes képernyőelem visszahívott függvénye lekérdezheti az ablakhoz tartozó *"purpose"* mező tartalmát a *g_object_get_data()* függvény segítségével és így megtudhatja milyen célra szolgál az ablak.

3.3.2. Az általános képernyőelem

A GTK+ programkönyvtár az objektumorientált módszertan értelmében minden képernyőelemet a *GtkWidget* típus utódjaként hoz létre, ezért a programkönyvtár minden képernyőeleme kezelhető *GtkWidget* típusú objektumként. Nyilvánvaló, hogy a képernyőelemek kezelése során igen fontosak azok az általánosan használható függvények, amelyek a képernyőelemeket *GtkWidget* típusú objektumokként kezelik.

A GTK+ programkönyvtár a képernyőelemek megjelenítésére, elrejtésére és megsemmisítésére a következő függvényeket biztosítja.

void gtk_widget_destroy(GtkWidget *widget); A függvény segítségével a képernyőelemeket megsemmisíthetjük, a tárolásukra használt memóriaterületet felszabadíthatjuk. A legtöbb esetben csak az ablakokat kell megsemmisítenünk ennek a függvénynek a hívásával, mert az ablak megsemmisítése következtében a GTK+ programkönyvtár megsemmisíti az ablakon belül található összes képernyőelemet.

A függvény paramétere a megsemmisítendő képernyőelemet jelöli a memóriában.

`void gtk_widget_show(GtkWidget *elem);` A függvény segítségével gondoskodhatunk arról, hogy a képernyőelem megjelenjen a képernyőn, azaz hogy a GTK+ programkönyvtár megjelenítse a képernyőelemet a monitoron.

Azok a képernyőelemek, amelyeket nem jelenítünk meg egyszerűen nem látszanak a képernyőn, így a felhasználó nem látja és nem tudja kezelni őket. Tudnunk kell viszont, hogy a képernyőelemek csak akkor jelenhetnek meg a képernyőn, ha megjelennek azok a képernyőelemek, amelyek hordozzák őket, azaz ha látszanak az ablakok, keretek, dobozok, amelyekben elhelyeztük őket.

A függvény paramétere azt a – már létrehozott – képernyőelemet jelöli a memóriában, amelyet meg kívánunk jeleníteni.

`void gtk_widget_hide(GtkWidget *elem);` A függvény segítségével az adott képernyőelemet elrejtethetjük a képernyőről. Az elrejtett képernyőelem – és a benne található összes képernyőelem szintén – egyszerűen „eltűnik” a képernyőről, így azokat a felhasználó nem láthatja és nem használhatja.

A függvény azonban nem semmisíti meg a képernyőelemet, így azt később, a `gtk_widget_show()` függvénnyel változatlan formában újra megjeleníthetjük.

A függvény paramétere az elrejtendő képernyőelemet jelöli a memóriában.

`void gtk_widget_show_all(GtkWidget *elem);` A függvény segítségével a képernyőelemet elrejtethetjük. Ez a függvény annyiban különbözik a már bemutatott `gtk_widget_show()` függvénytől, hogy a megadott képernyőelemen belül található összes képernyőelemet egyenként megjeleníti, így azok akkor is megjelennek a képernyőn, ha előzőleg nem jelenítettük volna meg.

A függvény paramétere a megjelenítendő képernyőelemet jelöli a memóriában.

`void gtk_widget_hide_all(GtkWidget *elem);` A függvény segítségével a képernyőelemet elrejtethetjük. A függvény annyiban különbözik a `gtk_widget_hide()` függvénytől, hogy a képernyőelemen belül található összes képernyőelemet egyenként elrejti.

A függvény paramétere az elrejteni kívánt képernyőelemet jelöli a memóriában.

A Glade program által készített, ablakokat létrehozó függvények a képernyőelemeket egyenként megjelenítik a `gtk_widget_show()` függvény segítségével, de nem jelenítik meg magát az ablakot, ami az összes

képernyőelemet hordozza. Így az ablak és a benne található képernyőelemek nem jelennek meg, azonban ha az ablakot megjelenítjük a `gtk_widget_show()` függvénnyel, minden képernyőelem megjelenik. Ezt mutatja be a 181. oldalon található 29. példa.

A Glade használata során a képernyőelemek beállítására szolgáló *tulajdonságok* ablakban az *általános* fülre kattintva a *látható* felirattal ellátott kapcsolóval beállíthatjuk, hogy az adott képernyőelem kezdetben megjelenjen-e. Ha ezzel a kapcsolóval a képernyőelem megjelenítését kikapcsoljuk, a képernyőelem kezdetben nem jelenik meg, mert a Glade által készített függvény nem kapcsolja be a képernyőelem láthatóságát a `gtk_widget_show()` függvény hívásával.

A GTK+ programkönyvtár lehetőséget biztosít arra is, hogy az egyes képernyőelemeket érzéketlenné tegyünk. Az érzéketlen képernyőelemeket a felhasználó nem tudja használni, az ilyen képernyőelemek ugyanis nem működnek. Az érzéketlen képernyőelemek azonban nem rejtettek, azaz a képernyőn továbbra is láthatók. A programkönyvtár árnyékolt, szürkített rajzolattal jelzi ugyan a felhasználónak, hogy a képernyőelemet nem lehet használni, de az elemet nem távolítja el.

`void gtk_widget_set_sensitive(GtkWidget *elem, gboolean érzékeny);` A függvény segítségével a képernyőelemeket érzéketlenné tehetjük és az érzéketlen képernyőelemek normális megjelenítését és működését visszaállíthatjuk.

A függvény első paramétere a megváltoztatandó képernyőelemet jelöli a memóriában. A második paraméter meghatározza, hogy a képernyőelem érzékeny vagy tiltott legyen-e. Ha a második paraméter értéke `FALSE`, a képernyőelem használatát a függvény tiltja.

A Glade programban, a képernyőelemek beállítására használható *tulajdonságok* ablakában az *általános* lapon az *érzékeny* felirattal ellátott kapcsolóval beállíthatjuk, hogy a képernyőelem kezdetben használható legyen-e.

A GTK+ programkönyvtár igen fejlett módon kezeli az egyes képernyőelemek méretét. A programkönyvtár által létrehozott képernyőelemek mérete általában éppen akkora, amekkora a helyes megjelenéshez és a használathoz szükséges. Mivel a képernyőelemek méretét erőteljesen befolyásolja a felhasználó által beállított megjelenési stílus, a képernyő számára előírt betűméret és még jónéhány körülmény, a program készítésekor általában nem tudhatjuk, hogy mekkora méretben kell az egyes képernyőelemeket megjeleníteni.

Meghatározhatja és lekérdezheti viszont a képernyőelemek legkisebb méretét a programozó, ha úgy találja, hogy a képernyőelem alapértelmezett mérete túlságosan kicsiny. Erre a következő függvény használható.

`void gtk_widget_set_size_request(GtkWidget *elem, gint`

szélesség, gint magasság); A függvény segítségével beállíthatjuk, hogy a képernyőelem mekkora méretet „kérjen” a grafikus felülettől, azaz beállíthatjuk mekkora legyen a képernyőelem legkisebb mérete. A képernyőelem a megadott méretnél nagyobb is lehet, elég ha a felhasználó egyszerűen nagyobbra állítja az ablakot, amelyben a képernyőelem megjelenik.

A függvény első paramétere a képernyőelemet jelöli, a második a szélességet, a harmadik pedig a magasságot adja meg képpontban. Ha a második vagy a harmadik paraméter `-1`, akkor a szélesség vagy a magasság nem változik. Ha például a képernyőelemnek csak a szélességét akarjuk megváltoztatni, akkor magasságként `-1`-et adunk meg. Ez elég hasznos, hiszen sok képernyőelem tartalmaz szöveget, feliratot, így a magasságát nem szerencsés megváltoztatni.

void gtk_widget_get_size_request(GtkWidget *elem, gint *szélesség, gint *magasság); A függvény segítségével lekérdezhetjük, hogy a képernyőelem lehető legkisebb mérete hány képpont.

A függvény első paramétere a képernyőelemet jelöli a memóriában, a második és harmadik paramétere pedig a memóriaterületet, ahová a szélességet, illetve a magasságot a függvény elhelyezi. Ha a magasságra vagy a szélességre nincs szükségünk, egyszerűen `NULL` értékű mutatót kell átadnunk a megfelelő paraméter helyén.

A Glade program *tulajdonságok* ablakában az *általános* lapon megadhatjuk a képernyőelem legkisebb szélességét és magasságát is.

Az általános képernyőelemek kezelésére hasznosak lehetnek még a következő függvények is.

gboolean gtk_widget_activate(GtkWidget *elem); A függvény segítségével a képernyőelemet „aktiválhatjuk”, mesterségesen olyan állapotba hozhatjuk, mintha a felhasználó lenyomta volna az Enter billentyűt a képernyőelemen.

gboolean gtk_widget_is_focus(GtkWidget *elem); A függvény segítségével megállapíthatjuk, hogy a képernyőelem az ablakán belül rendelkezik-e a billentyűzetfigyelés jogával. Az a képernyőelem amelyik rendelkezik a billentyűzetfigyelés jogával a felhasználó által lenyomott billentyű kódját megkapja feldolgozásra – feltéve, hogy az ablak, amelyben helyet foglal éppen aktív.

A függvény paramétere a vizsgálni kívánt képernyőelemet jelöli a memóriában, a visszatérési értéke pedig megadja, hogy rendelkezik-e a billentyűzetfigyelés jogával.

`void gtk_widget_grab_focus(GtkWidget *elem);` A függvény segítségével megadhatjuk, hogy melyik képernyőelem kapja meg a billentyűzetfigyelés jogát.

Ahhoz, hogy a függvény paramétere által kijelölt képernyőelem valóban megkapja a felhasználó által lenyomott billentyűk kódját, a képernyőelemnek képesnek kell lennie fogadni őket. A legtöbb képernyőelem esetében ez a viselkedés alapértelmezés szerint ésszerűen van beállítva – a beviteli mezők például képesek fogadni a billentyűkódokat, a keretek azonban nem –, de ezt a tulajdonságot a Glade *tulajdonságok* ablakában is beállíthatjuk a *fókuszban lehet* kapcsolóval.

A különféle képernyőelemek megjelenése függhet attól, hogy „fókuszban” vannak-e, de ez a használt stílustól is függ.

`void gtk_widget_set_name(GtkWidget *elem, const gchar *név);` A függvény segítségével beállíthatjuk a képernyőelem nevét. A névnek elsősorban a megjelenés szempontjából van szerepe, a felhasználó ugyanis beállíthatja, hogy az egyes nevekhez milyen megjelenés tartozzon. Általában nem szükséges, hogy ezt a függvényt használjuk.

`const gchar *gtk_widget_get_name(GtkWidget *elem);` A függvény segítségével lekérdezhethetjük, hogy a képernyőelemhez milyen név tartozik, így általánosan használható függvényeket készíthetünk, amelyek a képernyőelemeket nevüknek megfelelő módon kezelik.

4. fejezet

A G programkönyvtár

A G programkönyvtár jónéhány olyan függvényt és makrót is biztosít, amelyek kísértetiesen hasonlítanak a C programkönyvtár szabványos függvényeire, de nem pontosan egyeznek meg velük. Ha részletesen tanulmányozzuk ezeket az eszközöket, akkor látjuk, hogy a változtatásoknak egyértelműen a számítógépek fejlődése az oka. A G programkönyvtár általánosabb, összetettebb eszközöket biztosít, mint a C programkönyvtár, mivel a számítógépprocesszorok megnövekedett sebessége ma már lehetővé teszi, hogy bonyolultabb függvényeket írjunk, a mai számítógépek operatív memóriájának mérete lehetővé teszi, hogy nagyobb memóriaterületeket foglaljunk és így tovább. A G programkönyvtár sok függvényét azért érdemes használni, mert azok C programkönyvtárnál megszokott bosszantó apróságoktól mentesek.

A G programkönyvtár igen kiterjedt, sok makrót és függvényt biztosít az alkalmazásprogramozó számára. Mi a következő oldalakon csak a legfontosabbakat, a legáltalánosabban használt eszközöket mutatjuk be. Az érdeklődő olvasó a G programkönyvtár dokumentációjában naprakész információkat kaphat a többi eszközről is.

4.1. A G programkönyvtár típusai

A G programkönyvtár bevezet néhány egyszerű típust, amelyet érdemes megismernünk, annál is inkább, mivel a G programkönyvtár és a GTK+ programkönyvtár függvényei egyaránt használják ezeket. A G programkönyvtár által bevezetett alaptípusok a következők:

gboolean Logikai értéket hordozó típus, amelynek értéke **TRUE** vagy **FALSE** lehet.

gpointer Típus nélküli mutató (**void ***). Amikor a **gpointer** típust használjuk a felületes szemlélőnek úgy tűnik, hogy nem mutatót, hanem valamilyen más értéket használunk, fontos azonban tudnunk, hogy a hívott függvények ilyen esetben követni tudják a mutatót és meg tudják változtatni a memória tartalmát az adott helyen. Ezzel az új típussal a cím szerinti paraméterátadáshoz hasonlóan kimenő paramétereket hozhatunk létre a függvények számára.

gconstpointer Típus nélküli állandó mutató (**const void ***), amelyet akkor használunk, ha a hívott függvény felé jelezni szeretnénk, hogy a mutató követésével elért memóriaterületet nem szabad módosítani.

gchar Ugyanaz, mint a C programozási nyelv **char** típusa, de a G programkönyvtár (és a GTK+ programkönyvtár) legtöbb függvénye ezt a típust használja, ezért szerencsés, ha mi is ezt használjuk.

Megfigyelhető, hogy a G programkönyvtár jónéhány egyszerű C típust „átnevez” amelyet az általunk készített alkalmazásban – kényelmi okokból – nekünk is követnünk kell.

guchar Ugyanaz, mint **unsigned char**.

gint Ugyanaz, mint a **int**.

guint Ugyanaz, mint a **unsigned int**.

gshort Ugyanaz, mint a **short**.

gushort Ugyanaz, mint az **unsigned short**.

glong Ugyanaz, mint a **long**.

gulong Ugyanaz, mint a **unsigned long**.

gint8 Előjeles egész típus, amely garantáltan pontosan 8 bites, értéke mindig -128 és 127 közé esik.

guint8 Előjel nélküli egész típus, amely garantáltan 8 bites, értéke mindig 0 és 255 közé esik.

gint16 Előjel nélküli 16 bites egész, amelynek értéke mindig -32768 és 32767 közé esik.

guint16 Előjel nélküli 16 bites egész, amelynek értéke mindig 0 és 65535 közé esik.

gint32 Előjeles 32 bites egész, amelynek értéke mindig -2147483648 és 2147483647 közé esik.

guint32 Előjel nélküli egész, amelynek értéke mindig 0 és 4294967295 közé esik.

gint64 Előjeles 64 bites egész, amelynek értéke mindig -9223372036854775808 és 9223372036854775807 közé esik. Ez a típus csak akkor elérhető, ha a C fordító támogatja a **long long int** típust, de mivel a C programozási nyelv új szabványai ezt kötelezővé teszik, a G programkönyvtár dokumentációja szerint minden rendszeren használhatjuk a **gint64** típust.

guint64 Előjel nélküli 64 bites egész, amelynek értéke mindig 0 és 18446744073709551615 közé esik. Ez igen nagy szám, ha nem elég, akkor viszont baj van.

gfloat Ugyanaz, mint a C programozási nyelv **float** típusa.

gdouble Ugyanaz, mint a **double** típus.

gsize Előjel nélküli 32 bites egész, amely a különféle egyszerű és összetett típusok méretének tárolására készült.

gssize Előjeles 32 bites egész, amely különféle egyszerű és összetett típusok méretének tárolására készült. Mivel ez a típus előjeles, képes negatív méretek tárolására is, vagyis nem csak méretek hanem különféle hibakódok tárolására is használható (hiszen nyilvánvaló, hogy semminek nem lehet negatív a mérete).

A különféle típusok által hordozott legnagyobb és legkisebb értékek makróként is elérhetők. A makrók neve logikusan következik a típusok nevéből: **G_MININT**, **G_MAXINT**, **G_MAXUINT**, **G_MINSHORT**, **G_MAXSHORT**, **G_MAXUSHORT**, **G_MINLONG**, **G_MAXLONG**, **G_MAXULONG**, **G_MININT8**, **G_MAXINT8**, **G_MAXUINT8**, **G_MININT16**, **G_MAXINT16**, **G_MAXUINT16**, **G_MININT32**, **G_MAXINT32**, **G_MAXUINT32**, **G_MININT64**, **G_MAXINT64**, **G_MAXUINT64**, **G_MAXSIZE**, **G_MINFLOAT**, **G_MAXFLOAT**, **G_MINDOUBLE**, **G_MAXDOUBLE**.

4.2. Az általános célú makrók

A G programkönyvtár általános célú makrói közül a legfontosabbak a következők.

G_OS_WIN32 Ez a makró akkor létezik, ha a programot a Windows rendszerek valamelyikén fordítjuk le, tehát a segítségével operációs rendszer függő programrészeket készíthetünk, olyan sorokat, amelyek csak Windows rendszeren kerülnek a programba, vagy csak Windows rendszeren nem kerülnek a programba (**#ifdef** és **#ifndef** előfeldolgozó parancsok).

76

G_OS_BEOS E makró csak akkor létezik, ha a programunkat BeOS operációs rendszeren fordítják le.

G_OS_UNIX Ez a makró csak akkor létezik, ha a programunkat a számos UNIX változat valamelyikén fordítják.

G_DIR_SEPARATOR Az a karakter, amely az adott operációs rendszeren a könyvtárnevek elválasztására szolgál ('/' vagy '\\').

G_DIR_SEPARATOR_S A könyvtárnevek elválasztására szolgáló betű karakterláncként ("/" vagy "\\").

G_IS_DIR_SEPARATOR(c) A makró igaz értéket ad vissza, ha a paraméterként átadott karakter az adott operációs rendszeren a könyvtárnevek elválasztására használatos.

TRUE A logikai *igaz* érték.

FALSE A logikai *hamis* érték.

MIN(a, b) Ez a makró a két szám közül a kisebbet adja vissza.

MAX(a, b) A makró a két szám közül a nagyobbbat adja vissza.

ABS(a) A szám abszolút értéke.

CLAMP(x, alacsony, magas) A makró a három szám közül az első értékét adja vissza, de csak akkor, ha az az alsó és a felső határ közé esik, különben magát az alsó, illetve felső határt (természetesen attól függően, hogy a szám melyik oldalon „lóg ki” a megadott tartományból).

4.3. A hibakereséshez használható eszközök

A G programkönyvtár jónéhány olyan eszközt is biztosít, amelyek segítik a programhibák felderítését és kijavítását. A következő felsorolás ezek közül az eszközök közül csak azokat mutatja be, amelyek használatát könnyű megtanulni és amelyeket az Interneten felelhető programokban sokszor használnak. Mindenképpen javasolható ezen eszközök használata, hiszen a segítségükkel viszonylag kevés munka befektetésével nagyon megkönnyíthetjük a hibakeresést a programjainkban.

g_assert(*kifejezés*) A programhibák felderítésére igen kitűnően használható ez a makró. A makró megvizsgálja, hogy a paraméterként megadott kifejezés igaz logikai értéket ad -e. Ha igen, a program futása zavartalanul folytatódik, ha nem, akkor egy hibaüzenet

jelenik meg a szabványos hibacsatornán és a program futása megszakad. A hibaüzenetben a makró pontos helye és maga a kifejezés is megjelenik, így azonnal láthatjuk, hogy mi okozta a problémát.

Ha a makró helyén a `G_DISABLE_ASSERT` makró létezik, a lefordított programból a hibaellenőrzés kimarad, sem a program méretét nem növeli, sem a program futását nem lassítja. A program végső, kipróbált változatában tehát érdemes létrehoznunk a `G_DISABLE_ASSERT` makró, hogy ezt a hibakereső eszközt kihagyjuk.

A `g_assert()` makró általában a függvények elején használjuk, hogy ellenőrizzük a paraméterek értékét, azaz megfogalmazzuk az általunk írt programrészek helyes futásának előfeltételeit. Ha a program megfelelő pontjain a megfelelő ellenőrzéseket elhelyezzük a hibakeresés sokkal egyszerűbbé válik, de természetesen csak akkor, ha nem a hiba jelentkezésekor kezdjük el beírni a függvények elejére az előfeltételek ellenőrzését, hanem akkor, amikor a függvényeket megalkotjuk.

`g_assert_not_reached()` A makró hibaüzenetet ír a szabványos hibacsatornára, majd megszakítja a program futását. A makró használata megegyezik a `g_assert()` makró használatával, azzal a különbséggel, hogy ennek a makrónak nincsen paramétere.

`g_return_if_fail(kifejezés)` A makró figyelmeztető üzenetet ír a szabványos hibacsatornára és befejezi a függvény végrehajtását a `return` segítségével, ha a megadott kifejezés hamis logikai értéket képvisel. Ezt a makró csak olyan függvényben használhatjuk, amelynek visszatérési típusa `void`.

`g_return_val_if_fail(kifejezés,érték)` A makró figyelmeztető üzenetet ír a szabványos hibacsatornára és befejezi a függvény végrehajtását a `return` segítségével, ha a megadott kifejezés hamis logikai értéket képvisel. A makró második paramétereként megadott kifejezés meghatározza a visszatérési értéket.

`g_return_if_reached()` A makró figyelmeztető üzenetet ír a szabványos hibacsatornára és befejezi a függvény végrehajtását.

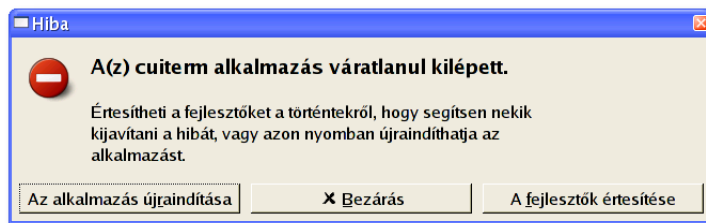
`g_return_val_if_reached(érték)` A makró figyelmeztető üzenetet ír a szabványos hibacsatornára és befejezi a függvény végrehajtását. A makró paramétere meghatározza a visszatérési értéket.

`g_message(formátumszöveg, ...)` A makró segítségével üzeneteket jeleníthetünk meg a szabványos hibacsatornán. A makró gondoskodik arról, hogy az üzeneteket azonnal megjelenjenek, így nem kell hívunk a `fflush()` függvényt.

A makró argumentumai a `printf()` könyvtári függvény argumentumaival egyeznek meg, az ott megszokott módon használhatjuk a formátumszöveget a kiírt szöveg formátumának meghatározására.

`g_warning(formátumszöveg, ...)` A makró a `g_message()` makróhoz hasonlóképpen működik, de nem egyszerű üzenetek, hanem figyelmeztetések kiírására használjuk. A makró a kiírt szöveg mellett jelzi azt is, hogy figyelmeztetésről van szó, amely nem kritikus ugyan, de hibás működésre utal.

`g_critical(formátumszöveg, ...)` A makró a `g_message()` és a `g_warning()` makróhoz hasonlóképpen működik, de kritikus hibaüzenetek kiírására használjuk.



4.1. ábra. A programhibát jelző ablak

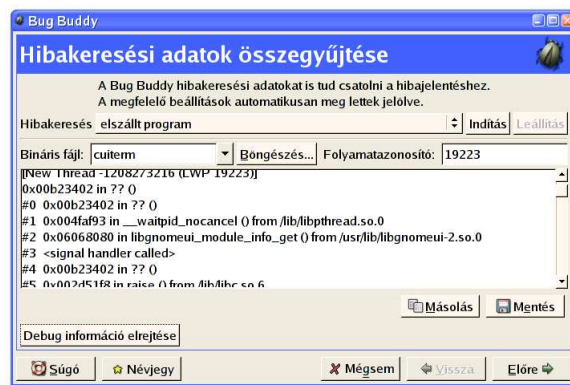
Amikor az alkalmazás futását megszakítjuk, a GTK+ programkönyvtár egy ablakot jelenít meg a képernyőn ennek jelzésére. Ezt az ablakot láthatjuk a 4.1. ábrán. Amint látjuk az ablakban kiválaszthatja a felhasználó, hogy bezárja-e a prog-

ramot, újraindítja azt vagy hibajelentést ír a fejlesztőknek.

Ha a programot bezárjuk a program egyszerűen kilép, ha azonban a programot újraindítjuk, a program ablaka újra megjelenik a képernyőn. Ez a lehetőség azonban komoly félreértéseket eredményezhet, ha a programot nem telepítettük a számítógépre. Amikor ugyanis a program újraindítását kezdeményezzük a bal szélső nyomógomb lenyomásával, a programunk telepített változata indul el. Ha tehát a programot a módosítás után nem telepítjük, hanem „helyben” indítjuk el, akkor a programhiba jelzése után már a telepített változat indul el. A tapasztalat azt mutatja, hogy általában ez a jelenség áll az „elsőször nem működik, aztán minden rendben van” jellegű hibák mögött.

Ha az ablak jobb oldali nyomógombjával a hibajelentés feladása mellett döntünk, megjeleníthetjük a hiba részletes leírását, ami egy külön ablakban tűnik fel. Ezt az ablakot mutatja be a 4.2. ábra.

A hiba részletes leírását adó ablak középső részén látható szöveges leírásban nem könnyű kiigazodni, de az ott olvasható információk nagymértékben megkönnyíthetik a hiba behatárolását. A listából megtudhatjuk, hogy milyen függvényhívások voltak érvényben a hiba bekövetkeztekor, azaz melyik függvények hívása okozta a hibát. Különösen hasznos



4.2. ábra. A programhiba részletes leírása

ez az információ akkor, ha a hibát nem a hibakereső makrók valamelyik váltotta ki, amikor a programhiba áratlanul következett be.

4.4. A dinamikus memóriakezelés

A G programkönyvtár néhány eszköze memóiafoglalásra és a foglalt memória felszabadítására szolgál. Fontos, hogy ismerjük ezeket a függvényeket, hiszen egyrészt egészen biztosan szükségünk lesz a programjainkba rájuk, másrészt viszont a G programkönyvtár és a GTK+ programkönyvtár függvényei is ezeket az eszközöket használják a dinamikus memóriakezelésre.

A C programozási nyelv szabványos programkönyvtárában található memóiafoglaló függvényekkel – `malloc()`, `realloc()` – szemben a G programkönyvtár hasonló célokra szolgáló függvényei – `g_malloc()`, `g_realloc()` – soha nem adnak vissza `NULL` értéket. Ha a memóiafoglalás nem jár sikerrel, a G programkönyvtár függvényei megszakítják a program futását, de nem adnak vissza `NULL` értékű mutatót. Ennek megfelelően nem szükséges a visszatérési értéket ellenőriznünk. (Léteznek viszont a `g_try_malloc()` és a `g_try_realloc()` függvények, amelyek probléma esetén szintén `NULL` értéket adnak vissza.)

`g_new(típus, darabszám)` A makró segítségével memóriát foglalhatunk adott típusú adatszerkezetek tárolására. A makró első paramétere az adott típus neve, a második paramétere pedig meghatározza, hogy hány darab elemet szeretnénk elhelyezni a lefoglalt memóriaterületen. A makró visszatérési értékének típusa az első paraméter által meghatározott típust jelölő mutató.

Nyilvánvaló, hogy ez a makró a C++ nyelv **new** kulcsszavához hasonlóan használható memóriaterületek foglalására. Mivel makróról van szó, a szerzőnek alkalma volt típust fogadni paraméterként (szöveg-szerű paraméterbehelyettesítés), ezért valóban kényelmesen használható eszközzel van szó.

g_new0(típus, darabszám) Ugyanaz, mint a **g_new()** makró, de a foglalt memóriaterületet 0 értékű bájtokkal tölti fel, mielőtt visszaadná a memóriacímet.

gpointer g_malloc(gulong méret); A függvény működésében hasonlít a **malloc()** könyvtári függvényhez, azaz a paraméterének megfelelő méretű memóriát lefoglalja, majd visszaadja az első foglalt memóriabájt címét, azonban – ahogyan azt már említettük – a **g_malloc()** soha nem ad vissza **NULL** értékű mutatót.

gpointer g_malloc0(gulong méret); Ugyanaz, mint a **g_malloc()**, de a lefoglalt memóriaterületet 0 értékű bájtokkal tölti ki.

gpointer g_realloc(gpointer memória, gulong méret); A függvény hasonlít a **malloc()** könyvtári függvényhez, azaz a megadott címen található dinamikusan foglalt memóriaterület méretét a megadott méretűre egészíti ki. A függvény által visszaadott mutató nem feltétlenül egyezik meg az első argumentumként megadott mutatóval, a függvény ugyanis szükség esetén átmásolja az alkalmazás adatterületét egy új memóriacímre.

A **g_realloc()** első argumentuma lehet **NULL** értékű is, ekkor a függvény feltételezi, hogy még nem foglaltunk memóriaterületet és így a viselkedése a **g_malloc()** függvény viselkedéséhez válik hasonlatossá.

void g_free(gpointer memória); A függvény segítségével a lefoglalt dinamikus memóriaterületet szabadíthatjuk fel. A **g_free()** a **free()** könyvtári függvénnyel szemben védett a **NULL** értékű mutatókkal szemben.

A bemutatott függvények segítségével a dinamikus memóriefoglalás és felszabadítás igen egyszerű és viszonylag biztonságos, ezért mindenképpen javasolható, hogy a C programkönyvtárak hasonló függvényei helyett ezeket használjuk.

4.5. A karakterláncok kezelése

A C programkönyvtár nem biztosít túl sok eszközt a karakterláncok kezelésére, ráadásul a biztosított függvényeknek is van néhány hiányossága.

A C programkönyvtár karakterláncokat kezelő függvényei nem védettek a `NULL` értékű mutatók ellen, azaz ha a függvényeknek ilyen mutatót adunk, a program futása azonnal megszakad. Ez – különösen kezdő programozók számára – meglehetősen nehézé és veszélyessé teheti a függvények használatát. A G programkönyvtár karakterláncok kezelésére alkalmas függvények ugyanakkor védettek a `NULL` értékű mutatók ellen. A védelem nyilvánvalóan lassítja a program futását, hiszen a hívott függvényeknek a paraméterként átadott mutatókat a használatuk előtt ellenőrizniük kell, de ma már a legtöbb esetben olyan gyors processzorokat használunk, hogy az a többletmunka nem okoz jelentős lassulást a program futásában.

A C programkönyvtár másik nagy hiányossága az, hogy a karakterláncokat kezelő függvények csak a 8 bites karakterábrázolást támogatják. A mai operációs rendszerek és alkalmazások a többnyelvű működés érdekében általában Unicode kódolást használnak, amelyet a C programkönyvtár nem támogat. A G programkönyvtár karakterláncokat kezelő függvényei az Unicode szabvány UTF-8 változatát használják a karakterábrázolásra. Az UTF-8 az Unicode legelterjedtebb változata, a segítségével könnyen kezelhetjük a magyar nyelv karaktereit is. Mivel maga a GTK+ programkönyvtár is ezeket a függvényeket használja, a GTK+ programkönyvtár segítségével készített grafikus felhatalmált felület maradéktalanul támogatja a magyar nyelvet.

A karakterláncokat kezelő függvények közül néhány dinamikusan foglalt memóriát használ. Ezek a függvények természetesen a `g_malloc()` függvénnyel foglalnak memóriaterületet, amelyet a `g_free()` függvénnyel szabadíthatunk fel.

4.5.1. A karakterláncok kezelése egyszerűen

A G programkönyvtár a karakterláncok kezelésére kétféle eszközt is biztosít. Az egyszerűbb eszköztár olyan függvényeket tartalmaz, amelyek a C programkönyvtárhoz hasonlóan egyszerű, 0 értékkel lezárt karaktersorozatokat használ a karakterláncok tárolására. Ezek a függvények paraméterként és visszatérési értéként `gchar` típusú mutatókat használnak. Az eszköztár legfontosabb függvényei a következők.

`gchar *g_strdup(const gchar *szöveg);` A függvény az `strdup()` könyvtári függvényhez hasonlóképpen működik, azaz lemásolja a paraméterként kapott karakterláncot egy dinamikusan foglalt memóriaterületre és visszaadja a lefoglalt memóriaterület címét.

A `g_strdup()` függvény azonban az `strdup()` függvénnyel ellentétben védett a `NULL` értékű mutatóval szemben, ha ilyen mutatót kap, a visszaadott mutató szintén `NULL` értéket lesz.

`gchar *g_strndup(const gchar *szöveg, gsize méret);` A függvény dinamikusan foglalt memóriaterületre másolja a megadott karakterláncnak legfeljebb a második paraméter által meghatározott számú karakterét. A függvény mindenképpen lezárja 0 karakterrel a visszaadott címen található karakterláncot, akkor is, ha emiatt a második paraméternél eggyel több memóriabájtot kell lefoglalnia.

A `g_strndup()` védett a `NULL` értékű mutatók ellen, ha ilyet kap a visszatérési érték szintén `NULL` értékű lesz.

`gchar *g_strnfill(gsize méret, gchar karakter);` A függvény a megfelelő méretű memóriaterületet foglalja, majd elhelyez benne egy adott méretű karakterláncot, amely egyforma karakterekből áll.

`gchar *g_stpcpy(gchar *cél, const char *forrás);` A függvény a forrásterületen található karakterláncot lemásolja a célterületre és visszaadja a célterületre másolt karakterláncot lezáró nulla értékű bájt címét, hogy a karakterláncot könnyen bővíthessük.

`gchar *g_strstr_len(const gchar *ebben, gssize hossz, const gchar *ezt);` A függvény az adott karakterláncban legfeljebb az adott hosszra keresi a második karakterláncot. A függvény visszatérési értéke a megtalált karakterlánc vagy `NULL`, ha a karakterlánc nem található.

`gchar *g_strrstr(const gchar *ebben, const gchar *ezt);` Karakterlánc keresése karakterláncban a karakterlánc végétől indulva. A függvény visszatérési értéke a megtalált karakterlánc első bájtjára mutat vagy `NULL`, ha a karakterlánc nem található.

`gchar *g_strrstr_len(const gchar *ebben, gssize méret, const gchar *ezt);` Karakterlánc keresése jobbról a méret korlátozásával.

`gboolean g_str_has_prefix(const gchar *szöveg, const gchar *előtag);` A függvény igaz értéket ad vissza, ha az első paraméterrel jelzett karakterlánc a második paraméterrel jelzett karakterlánccal kezdődik.

`gboolean g_str_has_suffix(const gchar *szöveg, const gchar *végződés);` A függvény igaz értéket ad vissza, ha az első paraméterrel jelzett karakterlánc a második paraméterrel jelzett karakterlánccal végződik.

`gchar *g_strdup_printf(const gchar *formátum, ...);` Ez a kintűnően használható függvény a `printf()` függvényhez hasonló paramétereket fogad és formázott karakterláncok kiírására használ-

ható. A `printf()` függvénnyel ellentétben azonban nem a szabványos kimenetre írja a függvény a karaktereket, hanem az általa foglalt memóriaterületre, amelyet a használat után fel kell szabadítanunk a `g_free()` függvény segítségével.

Különösen hasznos e függvény akkor, ha grafikus felülettel látjuk el a programunkat, hiszen ekkor az üzeneteket nem a szabványos kimenetre írjuk, hanem elhelyezzük a memóriában, hogy a grafikus felület függvényeivel megjeleníthessük valamelyik képernyőlemez belül.

Ha a programunkba csak a `glib.h` állományt töltöttük be – és nem töltöttük be például a `gtk.h` állományt –, szükséges lehet a `glib/gprintf.h` állomány betöltésére.

`gint g_printf(gchar const *formátum, ...);` A függvény a `printf()` szabványos könyvtári függvénnyel megegyező működésű.

`gint g_fprintf(FILE *fájl, gchar const *formátum, ...);` A függvény az `fprintf()` szabványos könyvtári függvénnyel megegyező működésű.

`gint g_sprintf(gchar *memória, gchar const *formátum, ...);` A függvény az `sprintf()` szabványos könyvtári függvénnyel megegyező működésű.

`gint g_snprintf(gchar *memória, gulong méret, gchar const *formátum, ...);` A függvény az `snprintf()` szabványos könyvtári függvénnyel megegyező működésű.

`gchar *g_strreverse(gchar *szöveg);` A függvény segítségével a karakterlánc bájttjainak sorrendjét az ellenkezőjére fordíthatjuk.

`gchar *g_strchug(gchar *szöveg);` A függvény segítségével a karakterlánc elején található szóközöket, tabulátor karaktereket és hasonló „fehér karaktereket” távolíthatjuk el. A függvény a megfelelő számú bájtal a memória eleje felé mozgatja a karakterláncot, de ami az elején található értéktelen karakterek számával rövidebb lesz. A függvény nem változtatja meg a karakterlánc tárolására esetleg lefoglalt dinamikus memória méretét.

A függvény paramétere a megváltoztatandó karakterláncot jelöli a memóriában, a visszatérési értéke pedig pontosan megegyezik a paraméterrel.

`gchar *g_strchomp(gchar *szöveg);` A függvény nagyon hasonlít a `g_strchug()` függvényhez, de nem a szöveg elején, hanem a végén található értéktelen karaktereket távolítja el.

`g_strstrip(szóveg)` Ez a makró a karakterlánc elején és végén található értéktelen karaktereket is eltávolítja a `g_strchug()` és a `g_strchomp()` függvények hívásával.

`gchar *g_strescape(const gchar *forrás, const gchar *kivételek);` A függvény segítségével a karakterláncban található különleges karaktereket az általánosan használt különleges jelölésre cserélhetjük le. A függvény az újsor karaktert a `\n` két-karakteres jelre cseréli, a tabulátor karaktert a `\t` jellel cseréli fel és így tovább. A függvény a karakterláncról másolatot készít egy dinamikusan foglalt memóriaterületre, hiszen a csere során a karakterlánc hossza növekedhet.

A függvény első paramétere a karakterláncot jelöli a memóriában, a második paramétere pedig egy karakterláncot, ami kivételként felsorolja azokat a karaktereket, amelyek különlegesek ugyan, mégis módosítás nélkül másolandók. A függvény visszatérési értéke a karakterlán másolatát tartalmazó dinamikusan foglalt memóriaterületet jelöli a memóriában.

`gchar *g_strcompress(const gchar *forrás);` A függvény segítségével a különleges jelöléssel ellátott karaktereket alakíthatjuk egyszerű formájukra. Ennek a függvénynek a működése a `g_strescape()` függvény működésének ellentettje.

A függvény első paramétere az átalakítandó karakterláncot jelöli a memóriában, a visszatérési értéke pedig a dinamikusan foglalt memóriaterületet jelöli, ahova a függvény az egyszerűsített karakterláncot elhelyezte.

`gchar *g_strcanon(gchar *szóveg, const gchar *érvényes_karakterek, gchar helyettesítő);` A függvény az első paraméterével jelzett karakterláncban kicseréli mindazokat a karaktereket, amelyek nem szerepelnek a második paraméterével jelzett karakterláncban a harmadik paraméterként átadott karakterre.

A függvény visszatérési értéke az első paraméterével egyezik meg. A függvény nem készít másolatot a karakterláncról, hiszen a karakterlánc hossza nem változik.

`gchar **g_strsplit(const gchar *szóveg, const gchar *határolószóveg, gint n);` A függvény feldarabolja az első paramétere által jelzett karakterláncot a második paramétere által jelzett karakterlánc mentén. A második paraméter által jelzett karakterlánc tehát a mezőelválasztó, amely lehet egy vagy több karakteres. Ha a második paraméter által jelzett karakterlánc

például a "\t ", akkor a mezőket egy tabulátor karakter és az utána következő szóköz választja el.

A függvény a darabolás során létrehozott részeket dinamikusan foglalt memóriaterületeken helyezi el, a címüket egy – szintén dinamikus memóriaterületen elhelyezett – tömbbe teszi, amelynek végét **NULL** értékű mutató jelöli. A visszatérési érték, a tömb címe tehát éppen olyan adatszerkezetet jelöl, amilyen a szabvány szerint a `main()` függvény második paramétere.

Ha a függvény első paramétere 0 hosszúságú karakterláncot jelöl, a visszatérési értéként átadott tömb első eleme **NULL** értékű, azaz az üres karakterláncot az üres tömbbel jelöljük.

A függvény harmadik paraméterével a feldarabolás után kapott részláncok számát korlátozhatjuk. Ha az elemek száma a harmadik paraméternél nagyobb lenne, az utolsó részlánc a teljes fennmaradó karakterláncot tartalmazni fogja. Ha nem akarunk korlátot megadni a részláncok számára nézve, harmadik paraméterként 1-nél kisebb értéket kell megadnunk.

A függvény visszatérési értéke által jelölt tömböt könnyedén végigjárhatjuk egyszerű ciklussal, megsemmisítésére – a szövegrészek és a tömb tárolására használt memóriaterület felszabadítására – pedig használhatjuk a `g_strfreev()` függvényt, amelynek formája a következő:

```
void g_strfreev(gchar **karakterlánc_tömb);
```

A függvény segítségével felszabadíthatjuk a dinamikus memóriaterületeket, amelyekre az első paraméterként átadott mutatóval jelölt tömb elemei mutatnak, valamint magának a tömbnek a tárolására használt, szintén dinamikusan foglalt memóriaterületet.

Fontos, hogy mind a tömbnek, mind az elemeivel jelölt memóriaterületeknek dinamikusan foglalt memóriaterületen kell elhelyezkedniük, ellenkező esetben a program futása azonnal megszakad.

```
gchar **g_strsplit_set(const gchar *szöveg, const gchar *határolójelek, gint n);
```

A függvény használata és működése megegyezik a `g_strsplit()` függvény használatával és működésével azzal a különbséggel, hogy ez a függvény mindig egy karakteres mezőelválasztó jeleket használ. A függvény második paramétere tehát olyan karakterláncokat jelöl, amely tartalmazza mindazokat a karaktereket, amelyeket határolójelként akarunk használni. Ha a második paraméter által jelzett karakterlánc például a "\t ", akkor a mezőket a tabulátor karakter vagy a szóköz választja el.

```
gchar *g_strconcat(const gchar *szöveg, ...);
```

E függvény segítségével karakterláncokat másolhatunk egymás után. A függvény

tetszőleges számú paramétert fogad, melyek azokat a karakterláncokat jelölik a memóriában, amelyeket egymás után akarunk másolni. Az utolsó paraméternek `NULL` értékűnek kell lennie, ellenkező esetben a létrehozott karakterlánc végén memóriaszemét jelenik meg és a program futása esetleg memóriahibával meg is szakad. A függvény visszatérési értéke egy dinamikusan foglalt memóriaterületet jelöl a memóriában, ami az egymás után másolt karakterláncokat tartalmazza.

`gchar *g_strjoin(const gchar *elvásztó, ...);` A függvény használata és működése megegyezik a `g_strconcat()` függvény használatával és működésével azzal a különbséggel, hogy ez a függvény az egymás után másolt karakterláncok közé elválasztójelként az első paraméter által jelölt karakterláncot másolja.

Fontos, hogy ennek a függvénynek is `NULL` értéket kell átadnunk utolsó paraméterként.

4.5.2. Az Unicode karakterláncok

A számítástechnika újkori történetére igen jellemző az Unicode szabvány elterjedt használata, ami lehetővé teszi, hogy a különféle nemzetek nyelveiben használt betűket és írásjeleket egységesen kezeljük. A magyar anyanyelvű felhasználóknak és programozóknak valószínűleg nem kell bizonygatni, hogy milyen fontos a nemzeti karakterkészletek egyszerű és szabványos kezelése, hiszen számukra a szabvány több évtizede megoldatlan problémára ad megnyugtató megoldást.

Az Unicode szabvány a karakterláncok ábrázolására háromféle megoldást ajánl. Az UCS32 változat minden karakter tárolására 32 bites kódszavakat használ, így a szöveg tárolására éppen négyszer akkora memóriaterületet használ, mint az ASCII kódolás. Az UTF-16 változat az európai kultúra által gyakran használt karakterek kódját 16 biten tárolja, a ritkább karakterek tárolására pedig 32 bitet. Ennek a változatnak a hátránya a bonyolultsága – változó hosszúságú kódszavakat nyilván bonyolultabb kezelni, mint az állandó hosszúságú kódszavakat – előnye pedig az, hogy ugyanaz a szöveg kisebb helyen elérhető.

Az Unicode szabvány harmadik változata a legelterjedtebb UTF-8, ami gyakoriságuktól függően 8, 16 vagy 32 bites kódszavakat rendel a karakterekhez. E változat nagy előnye – a takarékoság mellett – az, hogy az angol nyelvű szöveg karaktereihez éppen azokat a kódszavakat rendeli, amelyeket az ASCII kódolás is használ. Az angol nyelvű ASCII kódolású szöveg tehát egyben Unicode szöveg is.

A G programkönyvtár teljes mértékben támogatja az UTF-8 kódolást és az UTF-16 és UCS32 kódolása szövegeket átalakítására is biztosít eszközöket. A GTK+ programkönyvtár minden függvénye az UTF-8 kódolást

használja, így az magyar nyelvű szöveg kezelése nem jelenthet problémát – feltéve persze, hogy a programozó a megfelelő függvényeket használja a programjában.

Az UTF-8 kódolás kapcsán még azt érdemes megemlítenünk, hogy a GNU C fordító helyesen kezeli az ilyen kódolással készített karakterlánc-állandókat a programforrásban, ha tehát a szövegszerkesztő programunk ilyen kódolást használ, akár magyar nyelvű ékezetes szövegeket is elhelyezhetünk a kettős idézőjelek közt. Természetesen nem szerencsés ékezetes karaktereket használni a forrásprogramban, de lehetséges.

A G programkönyvtár az Unicode karakterek kezelésére a `gunichar` típust használja. A `gunichar` olyan 32 bites típus, ami bármelyik Unicode karakter tárolására használható, a 32 bites érték bizonyos kombinációja azonban nem megengedett, nem kódol érvényes Unicode karaktert. A G programkönyvtár `gunichar` típusának kezelésére a következő függvényeket használjuk elterjedten:

`gboolean g_unichar_validate(gunichar c);` A függvény igaz értéket ad vissza, ha a paraméterként átadott érték érvényes Unicode karakter.

`gboolean g_unichar_isalnum(gunichar c);` Igaz értéket ad vissza, ha a paraméterként átadott érték betűt vagy számot kódol.

`gboolean g_unichar_isalpha(gunichar c);` Igaz értéket ad vissza, ha a paraméterként átadott érték betűt kódol.

`gboolean g_unichar_iscntrl(gunichar c);` Igaz értéket ad vissza, ha a paraméterként átadott érték vezérlőkarakter kódja.

`gboolean g_unichar_isdigit(gunichar c);` Igaz értéket ad vissza, ha a paraméterként átadott érték számjegy kódja.

`gboolean g_unichar_isgraph(gunichar c);` Igaz értéket ad vissza, ha a paraméterként átadott érték olyan karakter kódja, ami nyomtatható, de nem szóköz.

`gboolean g_unichar_islower(gunichar c);` Igaz értéket ad vissza, ha a paraméterként átadott érték kisbetű kódja.

`gboolean g_unichar_isprint(gunichar c);` Igaz értéket ad vissza, ha a paraméterként átadott érték nyomtatható karakter kódja. Ez a függvény igaz értéket ad a szóköz kódjára.

`gboolean g_unichar_ispunct(gunichar c);` Igaz értéket ad vissza, ha a paraméterként átadott érték írásjel vagy más szimbólum kódja.

`gboolean g_unichar_isspace(gunichar c);` Igaz értéket ad vissza, ha a paraméterként átadott érték szóköz, tabulátor, sorvégjel vagy hasonló, helyet jelölő karakter kódja.

`gboolean g_unichar_isupper(gunichar c);` Igaz értéket ad vissza, ha a paraméterként átadott érték nagybetűt kódol.

`gboolean g_unichar_isxdigit(gunichar c);` Igaz értéket ad vissza, ha a paraméterként átadott érték a 16-os számrendszer számjegye.

`gboolean g_unichar_istitle(gunichar c);` igaz értéket ad vissza, ha a paraméterként átadott érték címszerű betű. (Az Unicode szabványban nem csak kis- és nagybetűk, hanem címszerű betűk is vannak. A címek szavait általában nagybetűvel kezdjük.)

`gboolean g_unichar_isdefined(gunichar c);` Igaz értéket ad vissza, ha a paraméterként átadott értékhez az Unicode szabvány valamilyen karaktert rendel, ha a paraméterként átadott érték nem „kihagyott” karakter.

`gboolean g_unichar_iswide(gunichar c);` Igaz értéket ad vissza, ha a paraméterként átadott érték olyan karaktert kódol, amely dupla szélességű helyet foglal.

`gunichar g_unichar_toupper(gunichar c);` A paraméterként kapott értéket nagybetűs formára alakítva adja vissza.

`gunichar g_unichar_tolower(gunichar c);` A paraméterként kapott értéket kisbetűs formára alakítva adja vissza.

`gunichar g_unichar_totitle(gunichar c);` A paraméterként átadott értéket címformára kódolva adja vissza.

`gint g_unichar_digit_value(gunichar c);` A függvény visszaadja, hogy a paraméterként kapott érték által kódolt karakter milyen értéket képvisel a tízes számrendszerben.

`gint g_unichar_xdigit_value(gunichar c);` Visszaadja, hogy a paraméterként kapott érték által kódolt karakter milyen értéket képvisel a tizenhatos számrendszerben.

`GUnicodeType g_unichar_type(gunichar c);` A függvény visszaadja, hogy a paraméterként kapott érték milyen jellegű jelentést hordoz az Unicode szabvány szerint. A függvény a következő kategóriákat különbözteti meg: `G_UNICODE_CONTROL`, `G_UNICODE_FORMAT`, `G_UNICODE_UNASSIGNED`, `G_UNICODE_PRIVATE_USE`, `G_UNICODE_SURROGATE`, `G_UNICODE_LOWERCASE_LETTER`, `G_UNICODE_MODIFIER_LETTER`, `G_UNICODE_OTHER_LETTER`,

```
G_UNICODE_TITLECASE_LETTER,    G_UNICODE_UPPERCASE_LET-
TER, G_UNICODE_COMBINING_MARK, G_UNICODE_ENCLOSING_MARK,
G_UNICODE_NON_SPACING_MARK,    G_UNICODE_DECIMAL_NUMBER,
G_UNICODE_LETTER_NUMBER, G_UNICODE_OTHER_NUMBER, G_UNI-
CODE_CONNECT_PUNCTUATION,    G_UNICODE_DASH_PUNCTUATION,
G_UNICODE_CLOSE_PUNCTUATION,    G_UNICODE_FINAL_PUNCTU-
ATION,    G_UNICODE_INITIAL_PUNCTUATION,    G_UNI-
CODE_OTHER_PUNCTUATION,    G_UNICODE_OPEN_PUNCTUATION,
G_UNICODE_CURRENCY_SYMBOL,    G_UNICODE_MODIFIER_SYMBOL,
G_UNICODE_MATH_SYMBOL,    G_UNICODE_OTHER_SYMBOL,    G_UNI-
CODE_LINE_SEPARATOR,    G_UNICODE_PARAGRAPH_SEPARATOR,
G_UNICODE_SPACE_SEPARATOR.
```

GUnicodeBreakType g_unichar_break_type(gunichar c);

A függvény visszaadja, hogy a paraméterként átadott érték az Unicode szabvány szerint milyen jellegű határolójel. A függvény a következő kategóriákat különbözteti meg: G_UNICODE_BREAK_MANDATORY, G_UNICODE_BREAK_CARRIAGE_RETURN, G_UNICODE_BREAK_LINE_FEED, G_UNICODE_BREAK_COMBINING_MARK, G_UNICODE_BREAK_SURROGATE, G_UNICODE_BREAK_ZERO_WIDTH_SPACE, G_UNICODE_BREAK_INSEPARABLE, G_UNICODE_BREAK_NON_BREAKING_GLUE, G_UNICODE_BREAK_CONTINGENT, G_UNICODE_BREAK_SPACE, G_UNICODE_BREAK_AFTER, G_UNICODE_BREAK_BEFORE, G_UNICODE_BREAK_BEFORE_AND_AFTER, G_UNICODE_BREAK_HYPHEN, G_UNICODE_BREAK_NON_STARTER, G_UNICODE_BREAK_OPEN_PUNCTUATION, G_UNICODE_BREAK_CLOSE_PUNCTUATION, G_UNICODE_BREAK_QUOTATION, G_UNICODE_BREAK_EXCLAMATION, G_UNICODE_BREAK_IDEOGRAPHIC, G_UNICODE_BREAK_NUMERIC, G_UNICODE_BREAK_INFIX_SEPARATOR, G_UNICODE_BREAK_SYMBOL, G_UNICODE_BREAK_ALPHABETIC, G_UNICODE_BREAK_PREFIX, G_UNICODE_BREAK_POSTFIX, G_UNICODE_BREAK_COMPLEX_CONTEXT, G_UNICODE_BREAK_AMBIGUOUS, G_UNICODE_BREAK_UNKNOWN, G_UNICODE_BREAK_NEXT_LINE, G_UNICODE_BREAK_WORD_JOINER, G_UNICODE_BREAK_HANGUL_L_JAMO, G_UNICODE_BREAK_HANGUL_V_JAMO, G_UNICODE_BREAK_HANGUL_T_JAMO, G_UNICODE_BREAK_HANGUL_LV_SYLLABLE, G_UNICODE_BREAK_HANGUL_LVT_SYLLABLE.

Az Unicode karakterláncok kezelésére a G programkönyvtár a szokásos, 0 karakterrel lezárt, karaktertömböt használja, paraméterként általában **gchar *** mutatót adva át a függvényeknek. A C nyelvben megszokott karakterláncok tehát Unicode karakterláncok is lehetnek, feltéve, hogy a megfelelő kódsorozatot használjuk. Mivel a G programkönyvtár

az UTF-8 kódolást részesíti előnyben, az ASCII karakterláncok egyben Unicode karakterláncok is.

Az Unicode karakterláncok kezelésére használt függvények a legtöbb esetben egy egész számot is fogadnak, ami megadja, hogy a karakterlánc hány bájtól áll. Ha ennek a paraméternek a helyén `-1`-et adunk át, a függvények a karakterlánc méretét a lezáró `0` értékű bájt alapján állapítják meg. Az Unicode karakterláncok kezelésére használatos függvények közül a legfontosabbak a következők:

`g_utf8_next_char(mutató);` E makró az UTF-8 karakterlánc következő elemének címét adja vissza. Mivel az UTF-8 változó hosszúságú kódszavakat használ a karakterláncban belül, ez a makró elengedhetetlenül fontos, ha a karaktereket végig akarjuk járni.

A makró paramétere egy mutató, ami UTF-8 karakterláncot jelöl, a visszatérési értéke pedig szintén egy mutató, ami a következő karakter elejére mutat. Nem szabad szem elől téveszteni, hogy az UTF-8 karakterláncban belül nem minden bájthatár jelöl új karaktert, azaz a karakterláncban belül található részláncok nem okvetlenül érvényes UTF-8 karakterláncok.

`gunichar g_utf8_get_char(const gchar *szöveg);` A függvény segítségével az UTF-8 karakterlánc első karakterének kódját kaphatjuk meg. A `g_utf8_next_char()` makró és a `g_utf8_get_char()` függvény segítségével az UTF-8 karakterlánc karaktereit bejárhatjuk.

A függvény paramétere a karakterláncot jelöli a memóriában, a visszatérési értéke pedig a karakterlánc első karakterének kódja.

`gunichar g_utf8_get_char_validated(const gchar *szöveg, gssize méret);` A függvény működésében és használatában hasonlít a `g_utf8_get_char()` függvényre, de azzal szemben fel van készítve hibás UTF-8 karakterláncok kezelésére is.

A függvény első paramétere UTF-8 karakterláncok jelöl a memóriában, második paramétere pedig a karakterlánc hosszát adja meg (vagy `-1` a `0` értékkel lezárt karakterlánc esetén).

A függvény visszatérési értéke a karakterlánc első karakterének kódja vagy negatív hibakód, ha a karakterlánc elején nem található érvényes UTF-8 karakter.

`gchar *g_utf8_offset_to_pointer(const gchar *szöveg, glong n);` A függvény segítségével megkereshetjük, hogy az UTF-8 karakterlánc adott sorszámú karaktere pontosan milyen memóriacímen kezdődik.

A függvény első paramétere UTF-8 karakterláncot jelöl a memóriában, második paramétere pedig megadja, hogy hányadik karaktert keressük. A függvény visszatérési értéke egy mutató, ami a keresett sorszámú karakter elejét jelöli a memóriában.

A függvény a G programkönyvtár újabb változataiban (2.10) a második paraméterként negatív értéket is képes elfogadni, azaz a segítségével visszafelé is haladhatunk a szövegben.

`glong g_utf8_pointer_to_offset(const gchar *szöveg, const gchar *karakter);` A függvény segítségével megállapíthatjuk, hogy az UTF-8 karakterlánc adott része hányadik karakterhelyre esik.

A függvény első paramétere UTF-8 karakterláncot jelöl a memóriában, a második paramétere pedig a karakterlánc valamelyik karakterének elejére mutat. A függvény visszatérési értéke megadja, hogy a második paraméterként átadott mutató hányadik karakterre mutat a karakterláncon belül.

A G programkönyvtár újabb változata (2.10) lehetővé teszi, hogy a második paraméter kisebb legyen, mint az első paraméter. Ilyenkor a visszatérési érték negatív.

`gchar *g_utf8_prev_char(const gchar *karakter);` A függvény segítségével megállapíthatjuk, hogy az UTF-8 karakterláncon hol található az előző karakter.

A függvény paramétere egy UTF-8 karakterlánc belsejébe mutat – de nem feltétlenül kell karakter elejére mutatnia –, visszatérési értéke pedig az előző karakter elejét jelzi a memóriában.

A függvénynek természetesen nem áll módjában leellenőrizni, hogy a paraméterként átadott mutató előtt folytatódik-e a karakterlánc, ezért, ha nem vagyunk óvatosak könnyen idegen memóriaterületre tévedhetünk.

`gchar *g_utf8_find_prev_char(const gchar *szöveg_eleje, const gchar *karakter);` A függvény működése és használata hasonlít a `g_utf8_prev_char()` függvényre, de e függvény képes leellenőrizni, hogy a karakterláncon van-e az adott karakterlánc előtt karakter.

A függvény első paramétere UTF-8 karakterláncot jelöl a memóriában, második paramétere pedig a karakterláncon belülre mutat, de nem feltétlenül kell karakter elejére mutatnia. A függvény visszatérési értéke a második paraméter által jelölt karakter előtti karakterre mutat a memóriában vagy `NULL` értékű, ha nincs ilyen karakter (mert már a karakterlánc elejére értünk).

`gchar *g_utf8_find_next_char(const gchar *karakter, const gchar *szöveg_vége);` A függvény segítségével megkereshetjük az UTF-8 karakterlánc következő karakterét.

A függvény első paramétere az UTF-8 karakterlánc valamelyik karakterére mutat – bár nem okvetlenül kell a karakter első bájtjára mutatnia –, második paramétere pedig a karakterlánc végét jelöli. A függvény visszatérési értéke az első paraméter által jelölt karakterlánc utáni karakter első bájtját jelöli vagy `NULL` érték, ha már nem található több karakter.

`glong g_utf8_strlen(const gchar *szöveg, gssize méret);` A függvény segítségével megállapíthatjuk, hogy az UTF-8 karakterlánc hány karakterből áll.

A függvény első paramétere UTF-8 karakterláncot jelöl a memóriában, második paramétere pedig megadja, hogy a karakterlánc hány bájtnyi memóriaterületet foglal el vagy `-1` a `0` értékkel lezárt karakterlánc esetében. A függvény visszatérési értéke megadja, hogy a karakterláncban hány karakter található.

`gchar *g_utf8_strncpy(gchar *cél, const gchar *forrás, gsize n);` A függvény az `strncpy()` szabványos könyvtári függvényhez hasonlóan legfeljebb adott számú karakter másolására használható, de azzal ellentétben az UTF-8 karakterláncban található karakterek számát veszi figyelembe.

`gchar *g_utf8_strchr(const gchar *szöveg, gssize méret, gunichar karakter);` A függvény a karakterláncban karaktert kereső `strchr()` szabványos könyvtári függvény UTF-8 karakterláncok kezelésére továbbfejlesztett változata.

A függvény első paramétere UTF-8 karakterláncot jelöl a memóriában, második paramétere pedig a karakterlánc méretét adja meg bájtban vagy `-1` a `0` értékkel lezárt karakterláncok esetében. A függvény harmadik paramétere megadja, hogy milyen karaktert keresünk a karakterláncban.

A függvény visszatérési értéke a keresett karakter első előfordulására mutat a karakterláncban vagy `NULL` érték, ha a karakter nem található.

`gchar *g_utf8_strrchr(const gchar *szöveg, gssize méret, gunichar karakter);` A függvény működése megegyezik a `g_utf8_strchr()` függvény működésével, de azzal szemben a karakter utolsó előfordulását jelölő mutatót adja vissza, azaz jobbról balra keres.

gchar *g_utf8_strreverse(const gchar *szöveg, gssize méret); A függvény segítségével az UTF-8 karakterláncban található karakterek sorrendjét ellentétére változtathatjuk.

A függvény első paramétere UTF-8 karakterláncot jelöl a memóriában, második paramétere pedig a karakterlánc hosszát adja meg bájtban vagy -1 a 0 értékkel lezárt karakterlánc használata esetén.

A függvény visszatérési értéke egy dinamikusan lefoglalt memóriaterületre mutat, ahol a karakterlánc található mégpedig megfordítva.

gboolean g_utf8_validate(const gchar *szöveg, gssize méret, const gchar **vége); A függvény segítségével megállapíthatjuk, hogy az adott karakterlánc érvényes UTF-8 karakterlánc-e, követi-e a szabványt. Mivel a G és a GTK+ programkönyvtárak sok függvénye, köztük az itt felsorolt függvények is feltételezik, hogy a kezelt adatterület érvényes UTF-8 karakterláncot tartalmaz, ez a függvény igen fontos lehet a munkánk során. Általában elmondhatjuk, hogy minden külső forrásból származó karakterláncot legalább egyszer ellenőriznünk kell ezzel a függvénnyel.

A függvény első paramétere az ellenőrizni kívánt karakterlánc elejét jelöli a memóriában, a második paramétere pedig megadja, hogy hány bájtnyi helyet foglal a karakterlánc a memóriában, esetleg -1 a 0 értékkel lezárt karakterlánc használata esetén.

A függvény harmadik paramétere egy mutatót jelöl a memóriában, ahova a függvény elhelyezi a karakterlánc végét jelző mutatót. Ha a karakterláncban olyan rész található, ami nem teljesíti a szabványt, a függvény a harmadik paraméterrel jelzett mutatóba a balról számított első érvénytelen karakter címét helyezi el.

A függvény visszatérési értéke **TRUE**, ha a karakterlánc csak érvényes UTF-8 karaktereket tartalmaz, **FALSE**, ha a függvény hibát talált. Szintén **FALSE** értéket ad vissza a függvény, ha a karakterlánc hosszát megadtuk, de a függvény a karakterláncot lezáró 0 értéket még a jelzett hossz elérése előtt megtalálta.

gchar *g_utf8_strup(const gchar *szöveg, gssize méret); A függvény segítségével az UTF-8 karakterláncot nagybetűs formára alakíthatjuk.

A függvény első paramétere az UTF-8 karakterláncot jelöli a memóriában, második paramétere pedig megadja a karakterlánc méretét bájtban vagy -1 a 0 értékkel lezárt karakterláncok esetében. A függvény visszatérési értéke dinamikus foglalt memóriaterületre mutat, ahol megtalálhatjuk a karakterlánc nagybetűs formára alakított másolatát.

```
gchar *g_utf8_strdown(const gchar *szöveg, gssize méret);
```

A függvény segítségével az UTF-8 karakterláncot kisbetűs formára alakíthatjuk.

A függvény első paramétere az UTF-8 karakterláncot jelöli a memóriában, második paramétere pedig megadja a karakterlánc méretét bájtban vagy -1 a 0 értékkel lezárt karakterláncok esetében. A függvény visszatérési értéke dinamikusan foglalt memóriaterületet jelöl a memóriában, ahol a szöveg kisbetűs formára alakított másolatát találhatjuk.

```
gchar *g_utf8_casefold(const gchar *szöveg, gssize méret);
```

A függvény segítségével az UTF-8 karakterláncot olyan formára alakíthatjuk, ami független a szövegen belüli kis- és nagybetűk használatától. Erre a műveletre általában azért van szükségünk mert a karakterláncokat a kis- és nagybetűk közti különbség figyelmen kívül hagyásával akarjuk összehasonlítani (esetleg sorba rakni, ami végső soron ugyanaz). Ilyenkor mindkét karakterláncot átalakítjuk e függvénnyel, majd az összehasonlítást elvégezzük a megfelelő függvénnyel (használhatjuk például a `g_utf8_collate()` függvényt.)

A függvény első paramétere az UTF-8 karakterláncot jelöli a memóriában, második paramétere pedig megadja a karakterlánc méretét bájtban vagy -1 a 0 értékkel lezárt karakterláncok esetében. A függvény visszatérési értéke dinamikusan foglalt memóriaterületet jelöl a memóriában a karakterlánc átalakított másolatával.

```
gchar *g_utf8_normalize(const gchar *szöveg, gssize méret, GNormalizeMode üzemmód);
```

Az Unicode szabvány szerint ugyanazt a karakterláncot több formában is ábrázolhatjuk. E függvény segítségével a karakterláncot alapformájára, egyszerű megjelenési formájára alakíthatjuk. A G programkönyvtár dokumentációja szerint az összehasonlítások, rendezések és egyéb fontos feldolgozási lépések előtt érdemes a karakterláncokat alapformájukra hozni, hogy az ábrázolás különbségei ne befolyásolják a feldolgozást.

A függvény első paramétere az UTF-8 karakterláncot jelöli a memóriában, a második paramétere pedig megadja a karakterlánc méretét bájtban, esetleg -1 a 0 értékkel lezárt karakterlánc használata esetén. A függvény harmadik paramétere megadja, hogy az átalakítást hogyan akarjuk elvégezni. Itt a következő állandók egyikét használhatjuk:

G_NORMALIZE_DEFAULT Ennek az állandónak a hatására a függvény az alapformára alakítást úgy végzi el, hogy az lehetőleg ne

módosítsa magát a szöveget. A szöveget ebben az üzemmódban a függvény a lehető legjobban elemeire bontott kódszavakkal tárolja.

G_NORMALIZE_DEFAULT_COMPOSE Ugyanaz mint az előző, de ebben az üzemmódban a függvény a lehető legösszetettebb kódokkal tárolja a szöveget.

G_NORMALIZE_ALL Ennek az állandónak a hatására a függvény erős átalakításokat is végez, amelynek hatására a formázásra vonatkozó információk elveszhetnek.

G_NORMALIZE_ALL_COMPOSE Ugyanaz mint az előző, de a függvény a lehető legösszetettebb kódokat használja.

Az összetett és kevésbé összetett kódok kapcsán érdemes tudnunk, hogy az Unicode szabvány lehetővé teszi a karakterek egységbe foglalását és átalakítását, így az „á” karakter például tárolható saját kódjával (összetett kód) és az „a” betű, valamint a hosszú ékezet összegeként (elemekre bontott kód). Az összetett kód rövidebb karakterláncot eredményez és könnyebb feldolgozni, az elemeire bontott kód pedig szisztematikusabb, szabályosabb.

A formázásra vonatkozóan érdemes tudnunk, hogy az Unicode ugyanannak a karakternek esetleg több formáját is támogathatja. A négyzet jelölésére például külön karaktert tartalmaz, ami a „²” formában „eleve felső indexben van”. Ha a karakterlánc alapformára hozása során az erős átalakítást választjuk a függvény minden karaktert az alapformával helyettesít (a példánk esetében ez a 2), így a formázás adta különbségek elvesznek.

Nyilvánvaló, hogy az átalakítás során a karakterlánc memóriában elfoglalt mérete megnövekedhet, így nem meglepő, hogy a függvény által visszatérési értéként adott mutató dinamikusan foglalt memóriaterületet jelöl a memóriában, ami a karakterlánc átalakított másolatát tartalmazza.

```
gint g_utf8_collate(const gchar *szöveg1, const gchar
*szöveg2);
```

A függvény két UTF-8 karakterlánc összehasonlítását végzi el. A függvény az összehasonlítás előtt a kis- és nagybetűk közti különbséget nem törli el a **g_utf8_casefold()** függvény segítségével (azaz a kis- és nagybetűk közti különbséget figyelembe veszi), de a karakterláncokat alapformára hozza a **G_NORMALIZE_ALL** üzemmódban.

A függvény paraméterei az összehasonlítandó karakterláncokat jelöli a memóriában, visszatérési értéke pedig negatív, ha az első karakterlánc kisebb, pozitív, ha nagyobb és 0, ha azonos a második karakterláncsal.

`gchar *g_utf8_collate_key(const gchar *szöveg, gssize méret);` A függvényt akkor használhatjuk, ha egy karakterláncot sok más karakterlánccal akarunk összehasonlítani. Ilyen esetben gyorsabb, ha a `g_utf8_collate()` ismételt hívása helyett e függvény segítségével átalakítjuk a karakterláncokat olyan formára, amely az egyszerű `strcmp()` szabványos könyvtári függvénnyel összehasonlítható. (A sebességnövekedés nyilván abból következik, hogy a sok karakterlánccal összehasonlítandó karakterláncot csak egyszer kell átalakítani.)

A függvény visszatérési értéke dinamikusan foglalt memóriaterületre mutat, ahol az első paraméterként átadott karakterlánc átalakított másolatát találhatjuk.

`gchar *g_utf8_collate_key_for_filename(const gchar *állománynév, gssize méret);` E függvény használata meg- egyezik a `g_utf8_collate_key()` működésével, a különbség csak annyi, hogy ez a függvény olyan szabályokat használ, ami állomá- névnevek esetében logikusabb sorrendet ad, mert különlegesen kezeli az állomá- névnevekben található „.” karaktert és a számokat.

Karakterláncok esetében elemi feladat, hogy a karakterlánc elemeit sorra végigjárjuk. Ezt az ASCII karakterláncok esetében egyszerűen meg- tehetjük, az UTF-8 karakterláncok esetében azonban kissé bonyolultabb a feladatunk. A következő példa bemutatja hogyan járhatunk végig egy UTF-8 karakterláncot a G programkönyvtár segítségével.

16. példa. A következő függvény a paraméterként kapott UTF-8 karak- terlánc karaktereit végigjárja és feldolgozza a G programkönyvtár segít- ségével.

```

1  gchar *
2  option_to_variable_name(gchar *option)
3  {
4      gunichar c;
5      GString *variable = g_string_new("OPTION");
6
7      do {
8          c = g_utf8_get_char(option);
9          option = g_utf8_next_char(option);
10         switch(c) {
11             case ' ':
12             case '-':
13                 g_string_append_unichar(variable, '_');
14                 break;
15             default:
```

```

16         g_string_append_unichar(variable, c);
17     }
18     } while (c != '\0');
19
20     return g_string_free(variable, FALSE);
21 }
```

A függvény a 8. sorban a `g_utf8_next_char()` függvény segítségével a soron következő karaktert olvassa a karakterlánc elejáról, a 9. sorban pedig a `g_utf8_next_char()` makró segítségével a következő karakterre lép. A példa 18. sorában megfigyelhetjük, hogy a karakterlánc végét a szokásos módon, a 0 érték figyelésével érzékelhetjük. (A példa további részeire a későbbiekben visszatérünk.)

4.5.3. A magas szintű karakterlánc-kezelés

A G programkönyvtár egy összetettebb eszköztárat is biztosít a karakterláncok kezelésére a `GString` típus segítségével. Ez az eszköztár sok, magas szintű szolgáltatást nyújtó függvényt tartalmaz amelyek képesek a karakterláncok méretének folyamatos nyomonkövetésére. A `GString` típusú karakterláncok mérete igény szerint növekszik, így igen könnyű a használatuk.

A `GString` struktúra a következő elemekből áll:

A <code>GString</code> struktúra	
<code>gchar *str</code>	A 0 értékű bájjal lezárt karakterlánc.
<code>gsize len</code>	A karakterlánc aktuális hossza.
<code>gsize allocated_len</code>	A foglalt memóriaterület hossza.

A struktúra `str` eleme egy olyan mutató, amely a szokásos formában – 0 bájjal lezárt karaktersorozatként – tárolja a karakterláncot. A G programkönyvtár minden függvénye garantálja, hogy a befejeződése után a karakterlánc a megfelelő módon le lesz zárva, így a karakterláncok az alacsonyszintű függvényekkel, sőt a C programkönyvtár függvényeivel is használhatjuk. A karakterlánc méretének növekedésekor az `str` mutató értéke megváltozhat, hiszen ha a karakterlánc bővítése az adott helyen nem lehetséges, akkor a bővítést végző függvénynek át kell helyeznie a karakterláncot más helyre.

A `GString len` mezője a karakterlánc hosszát adja meg bájtban mérve. Ebbe a hosszba nem számít bele a karakterláncot lezáró 0 bájt, ahogyan azt már a C programkönyvtár esetében is megszokhattuk.

A `GString` típusú karakterláncok kezelésére használható függvények közül a legfontosabbak a következők:

`GString *g_string_new(const gchar *kezdőérték);` A függvény segítségével új karakterláncot, új `GString` struktúrát hozhatunk létre.

A függvény paramétere a karakterlánc kezdeti értékét, visszatérési értéke pedig a létrehozott karakterláncot jelöli a memóriában.

`GString *g_string_new_len(const gchar *kezdőérték, gssize hossz);` A függvény segítségével új karakterláncot hozhatunk létre a kezdeti érték hosszának megadásával.

A függvény első paramétere a karakterlánc kezdőértékét jelöli a memóriában, második paramétere pedig a kezdőérték hosszát adja meg. Az első paraméter által jelölt memóriaterületen legalább annyi bájtnyi adatterületnek kell elérhetőnek lennie, amennyit a második paraméterrel előírtunk.

A függvény visszatérési értéke a létrehozott karakterláncot jelöli a memóriában.

`GString *g_string_sized_new(gsize kezdőhossz);` A függvény segítségével úgy hozhatunk létre karakterláncot, hogy megadjuk a kezdőhosszát, az a méretet, amelyet a létrehozáskor le szeretnénk foglalni a tárolására. A karakterlánc a későbbiekben hosszabb is lehet ennél, ez csak a létrehozáskor lefoglalt méret. Nyilván akkor lehet hasznos ez a függvény, ha a létrehozott karakterlánc a későbbiekben fokozatosan növekszik és a hatékonyabb memóriakezelés érdekében szeretnénk már a létrehozáskor gondoskodni a nagyobb memóriaterületről.

A függvény paramétere a lefoglalandó memóriaterület mérete, visszatérési értéke pedig a létrehozott karakterlánc helye a memóriában.

`GString *g_string_assign(GString *karakterlánc, const gchar *érték);` A függvény segítségével új értéket tárolhatunk a karakterláncban. A függvény a karakterlánc értékét megsemmisíti és új értéket helyez el benne.

A függvény első paramétere a megváltoztatandó karakterláncot, második paramétere az új értéket jelöli a memóriában.

A függvény visszatérési értéke az első paraméterrel egyezik meg.

`void g_string_printf(GString *karakterlánc, const gchar *formátum, ...);` E függvény segítségével a szabványos könyvtár `sprintf()` függvényéhez hasonlóan írathatunk szöveget a memóriában elhelyezett területre, de itt célterületként a `GString` adatszerkezetet használhatjuk, ami automatikusan akkora területet foglal, amekkorára a karakterlánc tárolásához szükséges.

A függvény első paramétere a karakterláncot jelöli a memóriában, ahol az eredmény el akarjuk helyezni. A karakterlánc eredeti tartalma törlődik.

A függvény második és további paramétereit jelentésükben megegyeznek a szabványos könyvtár `printf()` függvénycsaládjának paramétereivel.

`void g_string_append_printf(GString *karakterlánc, const gchar *formátum, ...);` E függvény működése megegyezik a `g_string_printf()` függvény működésével, azzal a különbséggel, hogy ez a függvény nem törli a karakterlánc eredeti tartalmát, hanem annak a végéhez írja hozzá a szöveget.

`GString *g_string_append(GString *karakterlánc, const gchar *szöveg);` E függvénnyel a karakterláncot a végéhez hozzáfűzött karakterekkel bővíthetjük.

A függvény első paramétere a bővítendő karakterláncot jelöli a memóriában, a második paramétere pedig a 0 értékkel lezárt szöveget, amelyet a karakterlánc végéhez kívánunk hozzáfűzni. A függvény visszatérési értéke az első paraméterrel egyezik meg.

`GString *g_string_append_c(GString *karakterlánc, gchar c);` A függvény segítségével egy karaktert másolhatunk a karakterlánc végére. Mivel ez a függvény nem támogatja az Unicode kódolást, helyette inkább a következő függvény ajánlható.

A függvény első paramétere a bővítendő karakterláncot jelöli a memóriában, második paramétere pedig a karakter, amellyel bővíteni kívánjuk a karakterláncot. A függvény visszatérési értéke az első paraméterrel egyezik meg.

`GString *g_string_append_unichar(GString *karakterlánc, guchar c);` Ennek a függvénynek a segítségével egy Unicode karaktert írhatunk a karakterlánc végére.

A függvény első paramétere a bővítendő karakterláncot jelöli a memóriában, második paramétere pedig megadja, hogy milyen Unicode karakterrel szeretnénk bővíteni a karakterláncot. A függvény visszatérési értéke az első paraméterrel egyezik meg.

`GString *g_string_prepend(GString *karakterlánc, const gchar *szöveg);` E függvény segítségével a karakterlánc elejét bővíthetjük úgy, hogy az eredeti karakterlánc az eredményül kapott karakterlánc végén jelenjen meg.

A függvény első paramétere a bővítendő karakterláncot, második paramétere pedig a 0 értékkel lezárt, a karakterlánc elejére máso-

landó szöveget jelöli a memóriában. A függvény visszatérési értéke az első paraméterével egyezik meg.

`GString *g_string_prepend_c(GString *karakterlánc, gchar c);` Ennek a függvénynek a működése megegyezik a `g_string_append_c()` függvény működésével, de nem a karakterlánc végére, hanem az elejére másolja a karaktert.

`GString *g_string_prepend_unichar(GString *karakterlánc, gunichar c);` Ennek a függvénynek a működése megegyezik a `g_string_append_unichar()` függvény működésével, de az Unichar karaktert nem a karakterlánc végére, hanem az elejére másolja.

`GString *g_string_insert(GString *karakterlánc, gssize hely, const gchar *szöveg);` A függvény segítségével a karakterlánc belsejébe másolhatunk szöveget úgy, hogy a bemásolt szöveg előtt és után az eredeti karakterlánc megmaradjon.

A függvény első paramétere a karakterláncot, harmadik paramétere pedig a 0 értékkel lezárt, bemásolandó szöveget jelöli a memóriában. A függvény második paramétere megadja, hogy az eredeti karakterlánc hányadik bájttól kezdődjön a bemásolt szöveg. A szöveg bájtszámának számozása 0-tól kezdődik, de a függvény nem támogatja az Unicode karaktereket, mivel az Unicode szövegben a karakterek nem minden esetben egy bájtnyi helyet foglalnak.

A függvény visszatérési értéke az első paraméterrel egyezik meg.

`GString *g_string_insert_c(GString *karakterlánc, gssize hely, gchar c);` E függvénnyel új karaktert szúrhatunk be a karakterláncba.

A függvény első paramétere a karakterláncot jelöli a memóriába, második paramétere pedig megadja, hogy hányadik bájthoz szeretnénk beszúrni az új karaktert. A függvény harmadik paramétere a beszúrandó karaktert adja meg, visszatérési értéke pedig megegyezik az első paraméterrel.

`GString *g_string_insert_unichar(GString *karakterlánc, gssize hely, gunichar c);` E függvény segítségével új Unicode karaktert szúrhatunk be a karakterláncba. A függvény a beszúrandó karaktert Unicode karakterként kezeli, de a karakterláncot egyszerű bájt sorozatként, így Unicode karakterláncok kezelésére nem igazán praktikus a használata.

A függvény paraméterei és visszatérési értéke a `g_string_insert_c()` függvény paramétereihez és visszatérési értékéhez hasonló.

`GString *g_string_erase(GString *karakterlánc, gssize hely, gssize hossz);` A függvény segítségével a karakterlánc megadott bájtoit törölhetjük.

A függvény első paramétere a módosítandó karakterláncot jelöli a memóriában, második paramétere megadja, hogy hányadik bájtól kívánjuk kezdeni a törlést, harmadik paramétere pedig azt, hogy hány bájtnyi szöveget akarunk eltávolítani. A bájtok számozása itt is 0-tól indul.

A függvény visszatérési értéke megegyezik az első paraméter értékével.

`GString *g_string_truncate(GString *karakterlánc, gsize hossz);` E függvény segítségével a karakterláncot megadott hosszúságúra vághatjuk.

A függvény első paramétere a karakterláncot jelöli a memóriában, második paramétere pedig megadja, hogy a karakterlánc végének levágása után hány bájtnyi szöveget tartalmazzon a karakterlánc (nem számolva a karakterláncot lezáró 0 értéket).

A visszatérési érték az első paraméterrel egyezik meg.

`gchar *g_string_free(GString *karakterlánc, gboolean felszabadít);` Ennek az egyébként igen hasznos függvénynek a segítségével a `GString` típusú karakterláncot megsemmisíthetjük, mégpedig úgy, hogy ha szükségünk van magára a szövegre, azt megtarthatjuk.

A függvény paramétere a megsemmisítendő karakterláncot jelöli a memóriában, második paramétere pedig megadja, hogy magát a szöveget is meg akarjuk-e semmisíteni.

Ha a második paraméter értéke `TRUE`, akkor a függvény magát a szöveget is megsemmisíti, a tárolására használt memóriaterületet felszabadítja. Ekkor a függvény visszatérési értéke `NULL`.

Ha a második paraméter értéke `FALSE`, a függvény a szöveg karaktereinek tárolására használt memóriaterületet nem szabadítja fel. Ekkor a függvény visszatérési értéke egy mutató, ami dinamikusan foglalt memóriaterületre mutat, ahol a 0 értékkel lezárt karakterláncot találhatjuk.

4.6. Az iterált adattípusok

A következő oldalakon olyan adatszerkezetekről olvashatunk, amelyek arra szolgálnak, hogy sok egyforma adatszerkezetet tartsanak nyilván a

memóriában. Ezeket az eszközöket általában iterált adatszerkezeteknek nevezzük.

Összetett programokban általában több listát, fát, keresőfát használunk a program által használt adatok kezelésére. A G programkönyvtár ebben hathatós segítséget tud nyújtani a programozó számára a következő oldalakon bemutatott adatszerkezetek, függvények segítségével.

4.6.1. A kétszeresen láncolt lista

A G programkönyvtár `GList` típusa kétszeresen láncolt lineáris lista megvalósítására használható. A `GList` struktúra szerkezete a következő:

A <code>GList</code> struktúra	
<code>gpointer data</code>	A listaelem adatterülete.
<code>GList *next</code>	A következő elem.
<code>GList *prev</code>	Az előző elem.

A struktúra bemutatott komponenseihez szabadon hozzáférhetünk és használhatjuk az értéküket.

A G programkönyvtár duplán láncolt listák kezelésére használatos függvényei közül a legfontosabbak a következők:

`GList *g_list_append(GList *lista, gpointer adat);`

A függvény segítségével új elemet szúrhatunk be a lista végére.

A függvény első paramétere a listát jelöli a memóriába. Ez a paraméter `NULL` értékű üres lista esetén.

A függvény második paramétere az új elem által hordozott adatokat jelöli a memóriában.

A függvény visszatérési értéke a lista új címe. Amikor a függvény segítségével új elemet helyezünk el egy üres listában – azaz a függvény első paramétere `NULL` értékű – a lista első elemének címe megváltozik, a visszatérési értéket tehát fel kell használnunk.

`GList *g_list_prepend(GList *lista, gpointer adat);` A függvény segítségével új elemet helyezhetünk el a lista elejére.

A függvény működésében és használatában megegyezik a `g_list_append()` függvény működésével és használatával, azzal a különbséggel, hogy a függvény mindig a lista elejére helyezi el az új elemet. A lista első elemének címe – amelyet a visszatérési érték ad meg – mindig megváltozik a függvény hívásának hatására.

```
GList *g_list_insert(GList *lista, gpointer adat, gint hely);
```

A függvény segítségével új elemet helyezhetünk el a listában adott sorszámú helyre.

A függvény első paramétere a listát jelöli a memóriában, második paramétere pedig az új listaelem által hordozott adatokat jelöli.

A függvény harmadik paramétere megadja, hogy az új listaelem hányadik helyre kerüljön a listán belül. Ha ez a szám negatív, vagy nagyobb mint a listában található elemek száma, az új elem a lista végére kerül.

A függvény visszatérési értéke a lista első elemének címe, amely megváltozhat a függvény hívásának hatására (nyilvánvalóan akkor, ha az első elem elé helyezünk el új elemet).

```
GList *g_list_insert_before(GList *lista, GList *következő, gpointer adat);
```

A függvény segítségével új elemet szúrhatunk be a listába adott elem elé.

A függvény első paramétere a listát – az első elemet – jelöli a memóriában, második paramétere annak az elemnek a címe, amely elé szeretnénk elhelyezni az új elemet, harmadik paramétere az új elem által hordozott adatot jelöli.

A függvény visszatérési értéke az első elem címe, amely megváltozhat, ha az első elem elé szúrunk be új elemet.

```
GList *g_list_insert_sorted(GList *lista, gpointer adat, GCompareFunc függvény);
```

A függvény segítségével új elemet helyezhetünk el a listában rendezett módon. Ha mindig ezzel a függvénnyel helyezünk el új elemeket a listában a lista mindvégig rendezett marad.

A függvény első paramétere a listát, második paramétere pedig az elhelyezendő új elem által hordozott adatot jelöli a memóriában.

A függvény harmadik paramétere annak a függvénynek a címe, amelyet a függvény az elemek összehasonlítására használhat, amely a sorrendet meghatározza. Ennek az összehasonlító függvénynek a működése meg kell, hogy egyezzen a `strcmp()` könyvtári függvénnyel, azaz 0 értéket kell visszaadnia, ha a két elem megegyezik, pozitív értéket kell visszaadnia, ha az első elem a nagyobb, illetve negatív értéket kell visszaadnia, ha a második elem a nagyobb. Az összehasonlító függvényt a `q_list_insert_sorted()` függvény a két összehasonlítandó elem adatterületét jelölő mutatóval hívja.

```
gint GCompareFunc(gconstpointer a, gconstpointer b);
```

Az elemek összehasonlítását végző függvény, amelyet nekünk kell elkészítenünk, ha rendezett listát akarunk létrehozni. A függvény paramétere a két összehasonlítandó adatterületet jelölik a memóriában, visszatérési értéke pedig az összehasonlítás eredményét adja meg.

A függvénynek 0 értéket kell visszaadnia, ha az első és második paraméter által jelzett területen található adat a sorrend szempontjából egyenértékű, pozitív értéket, ha az első nagyobb, illetve negatív értéket, ha az első kisebb mint a második.

```
GList *g_list_remove(GList *lista, gconstpointer adat);
```

A függvény segítségével adott adatot hordozó elemet távolíthatunk el a listából. A függvény első paramétere a listát, második paramétere az eltávolítandó elem által hordozott adatot jelöli a memóriában. A függvény mindig csak az első talált elemet távolítja el, a keresést az első találat után befejezi.

A függvény visszatérési értéke a lista első elemének címét adja meg, amely megváltozhat, ha éppen az első elemet távolítjuk el a listából.

A függvény használatának kapcsán fel kell hívnunk a figyelmet arra, hogy az nem a listában található adatterületek tartalmát hasonlítja össze a második paraméter által jelölt területen található adattartalommal, hanem a listaelemek adatterületeinek mutatóját hasonlítja össze a második paraméterrel. A függvénynek tehát nem „olyan” adatterületet kell jelölnie, amelyet el akarunk távolítani, hanem „azt” az adatterületet, amit el akarunk távolítani.

```
GList *g_list_delete_link(GList *lista, GList *elem);
```

A függvény segítségével adott elemet távolíthatunk el a listából.

A függvény első paramétere a listát, második paramétere pedig az eltávolítandó elemet jelöli a memóriában.

A lista visszatérési értéke a lista első elemének címe, amely megváltozhat, ha éppen az első elemet távolítjuk el a listából.

```
GList *g_list_remove_all(GList *lista, gconstpointer  
adat);
```

A függvény működése és használata megegyezik a `g_list_remove()` függvény működésével és használatával, azaz a különbséggel, hogy ez a függvény minden – adott adatterületet hordozó – elemet eltávolít, nem csak az elsőt.

```
void g_list_free(GList *lista);
```

A függvény megsemmisíti a listát, minden elemet eltávolít a listából.

A függvény paramétere a megsemmisítendő lista első elemét jelöli a memóriában.

Fontos tudnunk, hogy a függvény az egyes elemek által hordozott adatterületeket nem szabadítja fel, ha azokat dinamikus memória-foglalással vettük használatba magunknak kell gondoskodnunk a felszabadításukról mielőtt a függvényt hívnánk.

```
guint g_list_length(GList *lista);
```

A függvény megszámolja a listában található elemek számát. Az első paraméter a lista első elemének címét adja meg, a visszatérési érték pedig a lista hossza.

```
GList *g_list_reverse(GList *lista);
```

A függvény a lista elemeinek sorrendjét megfordítja.

A függvény első paramétere a listát jelöli a memóriában, a visszatérési értéke pedig az első elem címét adja meg, ami valószínűleg megváltozik a művelet hatására.

```
GList *g_list_sort(GList *lista, GCompareFunc függvény);
```

A függvény sorba rendezi a lista elemeit.

Az első paraméter a lista címe, a második paraméter pedig az elemek összehasonlítását végző függvény, amelyet már bemutattunk a `g_list_insert_sorted()` függvény kapcsán.

A függvény visszatérési értéke a lista első elemének címe, amely megváltozhat a művelet hatására.

```
GList *g_list_concat(GList *lista1, GList *lista2);
```

A függvény segítségével két listát egyesíthetünk. A függvény a második paraméter által jelölt listát az első paraméter által jelölt lista végére illeszti.

A függvény által visszaadott érték az egyesített listát jelöli a memóriában. A visszaadott érték valószínűleg – ha az első lista nem volt üres – megegyezik az első paraméterrel.

```
void g_list_foreach(GList *lista, GFunc függvény,  
gpointer paraméter);
```

A függvény bejáróciklust valósít meg, amelynek segítségével meghívhatunk egy általunk készített függvényt a lista minden elemével.

A függvény első paramétere a listát jelöli a memóriában, második paramétere pedig a meghívandó függvény címét adja meg.

A második paraméterrel meghatározott függvény két paramétert kap, az első az adott listaelem adatterületét jelöli, a második pedig

a `g_list_foreach()` függvénynek harmadik paraméterként megadott adatterület. A harmadik paraméter lehet `NULL` érték is.

```
void (*GFunc)(gpointer data, gpointer paraméter);
```

A függvényt nekünk kell megvalósítani, ha a `g_list_foreach()` függvénnyel minden listaelemet végig akarunk járni.

A függvény első paramétere az aktuális listaelem adatterületét jelöli, második paramétere pedig a `g_list_foreach()` számára átadott külön adatterületet jelöli.

```
GList *g_list_first(GList *elem);
```

A függvény a lista tetszőleges elemének felhasználásával megkeresi az első elemet.

```
GList *g_list_last(GList *elem);
```

A függvény a lista tetszőleges elemének felhasználásával megkeresi az első elemet.

```
GList *g_list_previous(GList *elem);
```

Ez az eszköz valójában makró, amely adott elem címéből előállítja a megelőző elem címét. Az első elem – amely előtt nincs újabb listaelem – esetében a makró `NULL` értéket ad vissza.

```
GList *g_list_next(GList *elem);
```

Ez a makró adott elem címéből előállítja a rákövetkező elem címét. Az utolsó elem esetében a makró `NULL` értéket ad vissza.

```
GList *g_list_nth(GList *lista, guint n);
```

Ez a függvény megkeresi az adott sorszámú elemet a listában.

A függvény első paramétere megadja a lista első elemének címét, a második pedig azt, hogy hányadik elemet keressük.

A függvény visszatérési értéke a keresett elemet jelöli a memóriában. Ha a keresett elem nem létezik – a lista rövidebb –, a visszatérési érték `NULL`.

```
gpointer g_list_nth_data(GList *lista, guint n);
```

A függvény hasonlít a `g_list_nth()` függvényre, de azzal ellentétben nem a keresett listaelemet, hanem a keresett listaelem által hordozott memóriaterületet jelöli a memóriában.

Ügyeljünk rá, hogy a függvény visszatérési értéke lehet `NULL`, ha a lista rövidebb annál, hogy a keresett elemet meg lehetne találni.

```
GList *g_list_find(GList *lista, gconstpointer adat);
```

A függvény segítségével kikereshetjük azt a listaelemet, amely a megadott adatterületet hordozza.

A függvény első paramétere a lista első elemének, második paramétere pedig a keresett adatterületnek a címét határozza meg.

A függvény visszatérési értéke a lista elemének címét adja meg, amely az adott adatterületet hordozza vagy `NULL`, ha az adatterület nem található.

Ügyelnünk kell arra, hogy a függvény nem hasonlítja össze a listaelemek adatterületeit a paraméterként megadott adatterülettel, csak a listaelemek adatterületeinek címét hasonlítja össze a megadott adatterület címével.

```
GList *g_list_find_custom(GList *lista, gconstpointer
    adat, GCompareFunc függvény);
```

A függvény segítségével adott adattartalmat hordozó listaelemet kereshetünk ki a listából.

A függvény első paramétere a lista első elemének címe, a második paramétere pedig a keresett adattartalmat jelöli a memóriában.

Ez a függvény összehasonlítja a lista egyes elemei által hordozott értékeket a megadott értékkel az összehasonlításra a harmadik paraméterként megadott függvényt használva fel. E függvény működését már bemutattunk a `g_list_insert_sorted()` függvény kapcsán.

A függvény visszatérési értéke a megtalált listaelem címe vagy `NULL`, ha az elem nem található.

```
gint g_list_position(GList *lista, GList *elem);
```

A függvény megadja, hogy az adott listaelem a lista hányadik eleme.

A függvény első paramétere a lista első elemének címe, második paramétere pedig a keresett listaelem címe.

A függvény visszatérési értéke a keresett elem sorszáma, vagy `-1`, ha a keresett elem nem található.

```
gint g_list_index(GList *lista, gconstpointer adat);
```

A függvény működése megegyezik a `g_list_position()` függvény működésével, azzal ellentétben azonban nem adott elemet, hanem adott memóriaterületet adatként hordozó elemet keres.

17. példa. A következő példaprogram a duplán láncolt lista használatát mutatja be. A program egy parancssoros felhasználói felület számára biztosítja a már begépelt és kiadott parancsok visszakeresésének (*command history*) lehetőségét.

108

```
1 static GList *history = NULL;
2 static GList *actual = NULL;
3
4 void
5 history_add_command(gchar *command)
6 {
7     history = g_list_append(history, g_strdup(command));
8     actual = g_list_last(history);
9 }
10
11 gchar *
12 history_get_prev_command(void)
13 {
14     gchar *command;
15
16     if (actual == NULL)
17         return NULL;
18     else
19         command = actual->data;
20
21     if (actual->prev != NULL)
22         actual = actual->prev;
23
24     return command;
25 }
26
27 gchar *
28 history_get_next_command(void)
29 {
30     gchar *command;
31
32     if (actual == NULL)
33         return NULL;
34     else
35         command = actual->data;
36
37     if (actual->next != NULL)
38         actual = actual->next;
39
40     return command;
41 }
42
43 gchar *
```

```

44 | history_get_first_command(void)
45 | {
46 |     actual = g_list_first(history);
47 |     return history_get_prev_command();
48 | }
49 |
50 | gchar *
51 | history_get_last_command(void)
52 | {
53 |     actual = g_list_last(history);
54 |     return history_get_next_command();
55 | }

```

A program 1. sorában egy változót hozunk létre, amely a lista első elemére fog mutatni. A listát ezzel a mutatóval tartjuk nyilván. A 2. sorban található mutató azt az aktuális elemet tartja nyilván, amelyet a felhasználó éppen használ a böngészés során.

A program 4–9. sorában láthatjuk azt a függvényt, amellyel új elemeket helyezünk el a listában. A 7. sorban látható értékadás a [g_list_append\(\)](#) függvény segítségével új elemet helyez el a lista végére. Figyeljük meg a függvény visszatérési értékének használatát és a függvénynek átadott, adattartalmat jelölő mutatót, amely ebben az esetben egyszerű karakterláncot jelöl a memóriában.

A 11–25. sorban található függvény segítségével az aktuális elem előtti, a 27–41. sorokban látható függvény segítségével pedig az aktuális elem utáni bejegyzés adatterületét kérdezhajjuk le. Figyeljük meg, hogy az adattartalom lekérdezése mellett a függvények 22., illetve a 38. sorokban módosítják az aktuális elemet kijelölő mutatót, ami azt eredményezi, hogy ezekkel a függvényekkel „végiglépkedhetünk” a lista elemein. Figyeljük meg azt is, hogy a 8. sorban az új elemet elhelyező függvény szintén módosítja az aktuális elemet, azaz, ha új elemet helyezünk el a listában, akkor egyben az aktuális elem a lista végére áll.

A 43–48., illetve az 50–55. sorokban látható függvényekkel a lista elejéről, illetve a végéről kérhetjük le az adattartalmat, egyben az aktuális elemet is mozgathatva.

4.6.2. A fa

A G programkönyvtár [GNode](#) típusa általános fák kezelését teszi lehetővé, ahol a fa egyes csomópontjainak tetszőleges számú leszármazottja lehet. A programkönyvtár a fa minden egyes csomópontjához egy mutatót rendel, amely az adott csomópont által hordozott adatokat jelöli a memóriában, így a [GNode](#) általánosan használható, rugalmas eszközként

110

tetszőleges konkrét feladat megoldására használható. (Ha a csomópontokhoz tartozó adatok gyors visszakeresése a cél, akkor lehetséges, hogy a 110. oldalon található 4.6.3. szakaszban bemutatott bináris keresőfára van szükségünk.)

A fa csomópontjainak adatait leíró **GNode** struktúra elemei a következők:

A GNode struktúra	
gpointer data	A csomóponthoz tartozó adatterület.
GNode *next	A csomópont melletti következő csomópont.
GNode *prev	A csomópont melletti előző csomópont.
GNode *parent	A csomópont szülője.
GNode *children	A csomópont első leszármazottja.

Amint láthatjuk a csomópont leszármazottait úgy tudjuk megkeresni, hogy a **children** mező tartalmából meghatározzuk az első leszármazott helyét, majd a leszármazottban található **next** mező segítségével végigjárjuk az egy szinten található csomópontokat.

4.6.3. A kiegyensúlyozott bináris keresőfa

A G programkönyvtár a **GTree** típus segítségével biztosítja a kiegyensúlyozott bináris keresőfát a programozó számára. A kiegyensúlyozott bináris keresőfa – amelyet sokszor egyszerűen keresőfának nevezünk – adatok gyors visszakeresését teszi lehetővé a memóriában elhelyezett adatszerkezetből.

A keresőfa minden csomópontja valamilyen összetett adatszerkezetet hordoz és minden keresőfa valamilyen keresési kulcs alapján teszi lehetővé az adatok gyors visszakeresését. A kulcs általában a hordozott összetett adatszerkezet valamelyik kitüntetett eleme, így a keresőfa összetett adatszerkezetek kitüntetett mezője alapján teszi lehetővé a visszakeresést. Nincs azonban semmi akadálya annak, hogy egy adatszerkezet egy időben több keresőfában szerepeljen, így a visszakeresést több kulcs, több mező alapján is lehetővé tehetjük.

A G programkönyvtár a kiegyensúlyozott bináris keresőfák kezelésére a következő függvényeket biztosítja.

GTree *g_tree_new(GCompareFunc függvény); A függvény segítségével új keresőfát hozhatunk létre.

A függvény paramétere azt a függvényt jelöli a memóriában, amelyet a keresőfa egyes elemeinek összehasonlítására használunk. Az összehasonlítást végző függvény megadása lehetővé teszi, hogy a keresőfát tetszőleges adattípusok, tetszőleges keresési kulcsok alapján kezeljük.

A függvény visszatérési értéke az új bináris keresőfát jelöli a memóriában.

`gint GCompareFunc(gconstpointer a, gconstpointer b);` Az elemek összehasonlítását végző függvény, amely a két összehasonlítandó kulcsot kapja paraméterként.

A függvény visszatérési értékének 0-nak kell lennie, ha a függvény a két kulcsot megegyezőnek ítéli, negatívnak, ha az első kulcs kisebb, mint a másik és pozitívnak, ha az első kulcs nagyobb mint a második.

`GTree *g_tree_new_with_data(GCompareDataFunc függvény, gpointer adat);` A függvény segítségével új bináris keresőfát hozhatunk létre. Azok a bináris keresőfák, amelyeket ezzel a függvénnyel hozunk létre annyiban különböznek a `g_tree_new()` függvénnyel létrehozott keresőfáktól, hogy az összehasonlítást végző függvény egy külön argumentumot is kap.

A függvény első paramétere a kulcsok összehasonlítását végző függvényt jelöli a memóriában, a második paraméter pedig megadja azt a mutatót, amelyet az összehasonlító függvény a kulcsok mellett paraméterként megkap.

A függvény visszatérési értéke az új bináris keresőfát jelöli a memóriában.

`gint GCompareDataFunc(gconstpointer a, gconstpointer b, gpointer adat);` Az elemek összehasonlítását végző függvény, amely a két összehasonlítandó kulcson kívül egy külön mutatót is kap paraméterként.

A függvény első és második paramétere a két összehasonlítandó kulcsot jelöli a memóriában, a harmadik paramétere pedig azt a memóriaterületet, amelyet a keresőfa létrehozásakor a `g_tree_new_with_data()` függvény második paramétereként megadtunk.

A függvénynek 0 értéket kell visszaadnia, ha a két kulcsot megegyezőnek ítéli, negatív értéket, ha az első kulcs értéke kisebb, pozitív értéket, ha az első kulcs értéke nagyobb, mint a második kulcs értéke.

`GTree *g_tree_new_full(GCompareDataFunc függvény, gpointer adat, GDestroyNotify kulcsfelszabadító, GDestroyNotify adatfelszabadító);` A függvény segítségével új bináris keresőfát hozhatunk létre úgy, hogy a keresőfában található adatalemeket felszabadító függvényeket is meghatározhatjuk.

A függvény egyes paraméterei és a visszatérési értéke a következők:

függvény A kulcsok összehasonlítását végző függvény címe a memóriában.

adat Az adatot jelöli a memóriában, amelyet a kulcsokat összehasonlító függvény paraméterként megkap.

kulcsfelszabadító A kulcsok tárolására szolgáló memóriaterületet felszabadító függvény címe a memóriában. A G programkönyvtár akkor hívja ezt a függvényt, amikor elemet távolítunk el a keresőfából és így felszabadítható az elemhez tartozó kulcs memóriaterülete.

Ha nem akarjuk használni ezt a szolgáltatást, akkor **NULL** értéket is megadhatunk.

adatfelszabadító Az adatterületek felszabadítására használt függvény címe. A G programkönyvtár akkor hívja ezt a függvényt, amikor elemet távolítunk el a keresőfából és ezért az elemhez tartozó adatterületet fel lehet szabadítani.

Ha nem akarjuk ezt a szolgáltatást használni **NULL** értéket is megadhatunk.

visszatérési_érték A függvény visszatérési értéke az új bináris keresőfát jelöli a memóriában.

void g_tree_insert(GTree *fa, gpointer kulcs, gpointer érték); A függvény segítségével új elemet szűrhatunk be a keresőfába. Az elem beszúrásához meg kell adnunk az elem és a kulcs értékét. Ha a megadott kulcs már létezik a fában, akkor a régi érték eltávolítása után az új érték kerül a fába, viszont a régi kulccsal.

A régi elem eltávolítása során, ha az érték felszabadításához megadtuk a függvényt – a **g_tree_new_full()** függvénnyel –, akkor a függvény hívja az érték memóriaterületének felszabadítását végző függvényt. Ha a fa létrehozásakor megadtuk a kulcs felszabadítását végző függvényt, akkor a felülírás során a függvény felszabadítja az új kulcsot.

Vegyük észre, hogy felülírásról csak akkor beszélhetünk, ha az új kulcs és a régi kulcs értéke az összehasonlítást végző függvény szerint megegyezik, de ez nyilvánvalóan nem jelenti azt, hogy a két kulcs ugyanazon a memóriaterületen található, hogy ugyanaz a memóriacímük.

A függvény első paramétere a már létrehozott bináris keresőfát, a második paramétere a kulcsot, a harmadik paramétere pedig a tárolandó értéket jelöli a memóriában.

void g_tree_replace(GTree *fa, gpointer kulcs, gpointer érték); A függvény segítségével új elemet helyezhetünk el a

keresőfában. A függvény a `g_tree_insert()` függvénytől abban különbözik, hogy ha a kulcs már létezik a fában, akkor ez a függvény az új kulcsot tartja meg és a régit szabadítja fel.

A függvény első paramétere a már létrehozott keresőfát, második paramétere a kulcsot, harmadik paramétere pedig a tárolandó értéket jelöli a memóriában.

`gint g_tree_nnodes(GTree *fa);` A függvény segítségével lekérdezhetjük, hogy a keresőfa hány kulcs/érték párt tartalmaz.

A függvény paramétere a keresőfát jelöli a memóriában, visszatérési értéke pedig megadja, hogy hány elem található a fában.

`gint g_tree_height(GTree *fa);` A függvény segítségével lekérdezhetjük a keresőfa magasságát, amely egyben azt is meghatározza, hogy legrosszabb esetben hány lépés kell a keresett elem megtalálásához. Az üres keresőfa magassága 0, az egy elemet tartalmazó keresőfa magassága 1 és így tovább.

A függvény paramétere a keresőfát jelöli a memóriában, a visszatérési értéke pedig megadja a fa magasságát.

`gpointer g_tree_lookup(GTree *fa, gconstpointer kulcs);` A függvény segítségével a fában a kulcs értéke alapján kereshetünk.

A függvény első paramétere a bináris keresőfát jelöli a memóriában, a második paramétere pedig a keresett kulcsértéket. A keresőfa létrehozásakor megadott összehasonlító függvény ezt a memóriaterületet fogja összehasonlítani a fában tárolt kulcsértékekkel, amíg a keresés egyezést nem talál vagy nem bizonyosodik róla, hogy a kulcsérték nem található a fában.

A függvény visszatérési értéke kulcshoz tartozó értéket jelöli a memóriában vagy `NULL`, ha a kulcsérték nem található.

`void g_tree_foreach(GTree *fa, GTraverseFunc függvény, gpointer adat);` A függvény segítségével bejárhatjuk a keresőfában tárolt elemeket. A függvény a kulcsok növekvő sorrendjében bejárja a keresőfa elemeit és a megadott mutató segítségével minden csomóponttal meghívja az általunk megadott függvényt.

A függvény paraméterei a következők:

fa A keresőfát jelöli a memóriában.

függvény A függvény, amelyet a keresőfa minden elemével meg akarunk hívni. Ez a függvény nem módosíthatja a keresőfa szerkezetét, nem helyezhet el új elemeket és nem távolíthat el elemeket.

adat Mutató, amelyet a második paraméterrel meghatározott függvénynek át szeretnénk adni paraméterként.

Ha a bejárást arra szeretnénk használni, hogy a segítségével bizonyos feltételnek megfelelő elemeket eltávolítsunk a keresőfából, akkor az egyező elemeket egy külön adatszerkezetben – például duplán láncolt listában – el kell helyeznünk, majd – amikor a `g_tree_foreach()` függvény visszatér – a listában szereplő elemeket el kell távolítanunk a keresőfából.

gboolean `GTraverseFunc(gpointer kulcs, gpointer érték, gpointer adat);` A keresőfa bejárása során meghívott függvény típusa. Ilyen típusú függvény mutatóját kell a `g_tree_foreach()` függvény második paramétereként átadnunk.

A függvény első és második paramétere az aktuális kulcsot és értéket jelöli a memóriában. A harmadik paraméter az a mutató, amelyet a bejárás indításakor megadtunk.

A függvénynek **TRUE** értéket kell visszaadnia ahhoz, hogy a bejárást megszakítsa. Ha a függvény **FALSE** értéket ad vissza a bejárás folytatódik.

void `g_tree_remove(GTree *fa, gconstpointer kulcs);` A függvény segítségével a keresőfában található elemet eltávolíthatjuk a kulcsérték ismeretében. Ha a fa létrehozásakor megadtuk az érték és/vagy kulcs felszabadítását végző függvényt, a `g_tree_remove()` meghívja ezeket is.

A függvény első paramétere a keresőfát, második paramétere pedig az eltávolítandó elem kulcsértékét jelöli a memóriában.

void `g_tree_destroy(GTree *fa);` A függvény segítségével a teljes keresőfát megsemmisíthetjük. Ha a fa létrehozásakor megadtuk a kulcs és/vagy elem felszabadítását végző függvényt a `g_tree_destroy()` minden elemre elvégzi a felszabadítást is.

A függvény paramétere a megsemmisítendő fát jelöli a memóriában.

4.7. Állománykezelés

A G programkönyvtár néhány kitűnő eszközt biztosít a programozó számára az állományok kezelésére. Az eszközök egy része az állománynevek operációs rendszertől független kezelésében nyújt hathatós segítséget, egy másik része pedig az eseményvezérelt programozásba jól beilleszthető, párhuzamos programozásra is felkészített eszköztárat biztosít az állománykezelés segítésére. A következő oldalakon ezekről az eszközökről olvashatunk.

4.7.1. Az állománynevek kezelése

A G programkönyvtár az állományok nevének és elérési útjuknak a kezelésére a következő – a használt operációs rendszertől függetlenül használható – függvényeket biztosítja.

gchar *g_build_path(const gchar *elvásztó, const gchar *név1, ...); A függvény segítségével a könyvtárbejegyzést elemekből építhetjük fel. Különösen alkalmas ez a függvény arra, hogy az elérési útból – ami esetleg több könyvtárnevet is tartalmaz – és az állomány nevéből előállítsuk a teljes elérési utat, ami alapján az állomány megnyithatjuk.

A függvény első paramétere a könyvtárak nevét az elérési útban elválasztó jelet tartalmazó karakterlánc. Itt kitűnően használhatjuk a már bemutatott **G_DIR_SEPARATOR_S** állandót.

A függvény további paraméterei azok a karakterláncok, amelyekből a teljes elérési utat elő akarjuk állítani. Az utolsó paraméternek **NULL** értékűnek kell lennie.

A függvény visszatérési értéke olyan dinamikusan foglalt memóriaterületre mutat, ahol a teljes állománynevet találjuk a szokásos módon 0 értékkel lezárva.

gchar *g_build_filename(const gchar *név1, ...); Ez a függvény nagyon hasonlít a **g_build_path()** függvényhez, de a használata közben nem kell – és nem is lehet – megadni az elválasztójelet, a függvény mindig az adott operációs rendszernek megfelelő jelet választja.

Nem szabad elfelejtenünk, hogy a függvény utolsó paraméterének **NULL** értékűnek kell lennie.

gchar *g_find_program_in_path(const gchar *programnév); A függvény segítségével az ösvény (*path*) által meghatározott helyeken kereshetünk futtatható programállományokat. Ez a függvény éppen úgy keresi a megadott állományt, mint azok az eszközök, amelyekkel a programokat elindíthatjuk, úgy is mondhatnánk, hogy ugyanazt a programot fogja „megtalálni” amelyiket elindítanánk.

A függvény paramétere a program neve, visszatérési értéke pedig a program teljes elérési útját tartalmazó dinamikusan foglalt memóriaterület, ha a program az ösvény valamelyik pontján megtalálható, **NULL**, ha nem.

gchar *g_path_get_basename(const gchar *teljes_név); E függvény segítségével az állomány teljes elérési útjából előállíthatjuk az állománynevet, az állomány nevének könyvtárnév nélküli részét.

A függvény paramétere az állomány teljes elérési útja, visszatérési értéke pedig olyan mutató, ami a dinamikus memóriaterületet jelöli a memóriában, ahol a függvény az állomány nevét elhelyezte.

`gchar *g_path_get_dirname(const gchar *teljes_név);` E függvény nagyon hasonlít a `g_path_get_basename()` függvényhez, de azzal ellentétben a teljes név könyvtárnév részét adja vissza, nem pedig az állománynév részét.

`gboolean g_path_is_absolute(const gchar *név);` E függvénnyel megállapíthatjuk, hogy az állománynév abszolút elérési utat tartalmaz-e.

A függvény paramétere az állomány nevét jelölő mutató, visszatérési értéke pedig igaz, ha a név abszolút elérési utat tartalmaz.

Gyakran előfordul, hogy a felhasználó saját könyvtárában található állományt szeretnénk megnyitni vagy más módon használni attól függetlenül, hogy a program munkakönyvtára éppen mi. Ilyenkor a legegyszerűbb, ha az állományra a teljes – más néven abszolút – elérési úttal hívatkozunk. Ezt mutatja be a következő példa.

18. példa. A következő programrészlet bemutatja hogyan állíthatjuk elő a felhasználó saját könyvtárában található állomány abszolút elérési útját a saját könyvtár nevének lekérdezésével.

```

1  gchar      *filename;
2
3
4  filename = g_build_path(G_DIR_SEPARATOR_S,
5                          g_getenv("HOME"),
6                          ".bash_history", NULL);
7
8  if (g_stat(filename, &stat_buf) != 0){
9      g_warning("Unable to stat file '%s'", filename);
10     g_free(filename);
11     return NULL;
12 }
```

A programrészletben az 5. sorban láthatjuk a `g_getenv()` függvény hívását. E függvény a környezeti változók lekérdezésére használható és mivel a hagyomány szerint a `HOME` környezeti változó a felhasználó saját könyvtárának abszolút elérési útját tartalmazza, a függvény a számunkra nagyon hasznos.

Az állomány teljes elérési útját a könyvtárnévből és az állomány nevéből a 4–6. sorokban állítjuk elő és a 8. sorban kezdjük használni.

Figyeljük meg a 10. sorban, hogy az állománynevet tároló területet a használat után fel kell szabadítanunk. A `g_getenv()` függvény azonban nem dinamikusan foglalt memóriaterület címét adja vissza, ezért ezt a területet nem is szabad felszabadítanunk.

4.8. A parancssori kapcsolók kezelése

A parancssorban a programnak átadott kapcsolók és paraméterek egyszerű felhasználói felületet biztosítanak a felhasználó számára, amellyel a program viselkedését, működését befolyásolhatja. A felhasználó számára nagyon fontos a parancssorban megadható kapcsolók használata még akkor is, ha a program egyébként rendelkezik grafikus felhasználói felülettel, de egyszerűbb programok esetén előfordulhat, hogy a program viselkedését egyedül a parancssori paraméterek segítségével lehet vezérelni. A G programkönyvtár néhány egyszerűen használható eszközt biztosít a parancssori kapcsolók értelmezésére. A következő oldalakon arról olvashatunk hogyan használhatjuk ezeket az eszközöket.

A programok kapcsolóira a kapcsolók rövid vagy hosszú nevével hivatkozhatunk. A rövid kapcsolónevek egyetlen betűből állnak (például `-h` vagy `-v`), amelyeket egybe is írhatunk (például `-hv`). A hosszú neveket mindig két kötőjellel vezetjük be (például `--help`) vagy `--version`. A hosszú neveket soha nem írhatjuk egybe. Nyilvánvaló, hogy a rövid neveket a gépelés egyszerűsítésére, a hosszú neveket pedig az olvashatóság érdekében használjuk, a rövid neveket tehát elsősorban az interaktív használat során, a hosszú neveket pedig a héjprogramok készítésekor részesítjük előnyben.

Bizonyos kapcsolók értéket is kaphatnak. Az értéket a rövid kapcsolók után szóközzel elválasztva írjuk (például `-O 5`), a hosszú nevek után pedig az egyenlőségjellel jelöljük (például `--optimization=5`). A kapcsolók számára szám jellegű, szöveges vagy állománynév jellegű értéket szokás megadni.

Fontos tudnunk, hogy a legtöbb program – így a G programkönyvtár segítségével készített programok is – feltételezik, hogy a programnak nem adunk kapcsolókat a különleges, `--` értékű argumentum után. Ez az eszköz teszi lehetővé, hogy a programnak olyan szöveges argumentumokat adjunk át, amelyek kötőjellel kezdődnek. Ha például az `rm` programmal a `-fájl` állományt akarjuk törölni, akkor az `rm -- -fájl` parancsot kell begépelnünk, mert az `rm` program feltételezi, hogy a `--` utáni argumentumok mindegyike állománynév.

Látható, hogy a parancssori argumentumok értelmezése nem egyszerű és ha be akarjuk tartani a „legkisebb meglepetés” elvét (amely igen fontos a felhasználók meglegedettségének szempontjából), szerencsés, ha az általunk készített alkalmazás a többi programhoz hasonlóan körülte-

118

kintő módon kezeli az argumentumait. Ebben nyújt komoly segítséget a G programkönyvtár néhány eszköze.

A programok kapcsolóinak tulajdonságait a G programkönyvtár a **GOptionEntry** struktúrában tárolja, amely a következő komponenseket tartalmazza:

A GOptionEntry struktúra	
const gchar *long_name	A kapcsoló hosszú neve.
gchar short_name	A kapcsoló rövid neve.
gint flags	A kapcsoló láthatósága.
GOptionArg arg	Az érték típusa.
gpointer arg_data	Az érték.
const gchar *description	A kapcsoló leírása.
const gchar *arg_description	Az argumentum jele.

A struktúra elemei a következő jelentéssel bírnak:

long_name A kapcsoló hosszú neve a **--** jelek nélkül.

short_name A kapcsoló rövid neve a **-** jel nélkül (egyetlen betű).

flags A kapcsoló néhány egyszerű tulajdonságát határozza meg ez az elem. Itt a következő állandókból bináris *vagy* kapcsolattal készített érték adható meg:

G_OPTION_FLAG_HIDDEN Ha ezt az bitet bekapcsoljuk a kapcsoló rejtett lesz, nem jelenik meg a **--help** kapcsoló hatására kiírt sűgőben.

G_OPTION_FLAG_IN_MAIN Ha ezt a bitet bekapcsoljuk a kapcsoló mindig a kapcsolók központi szakaszában szerepel, akkor is, ha más szakaszban hoztuk létre (e könyvben nem tárgyaljuk a kapcsolók szakaszokra osztását, az alpontok létrehozását).

G_OPTION_FLAG_REVERSE A ki- és bekapcsolható kapcsolóknál használatos ez a bit, amely arra utal, hogy a kapcsoló hatása ellenétes, ha használjuk a program adott szolgáltatását kikapcsoljuk.

Ha a bemutatott különleges tulajdonságokat nem kívánjuk megadni, akkor ennek a mezőnek a helyén használjuk a 0 értéket!

arg Ez a mező azt határozza meg, hogy a kapcsoló milyen értéket kaphat, milyen típusú értéket lehet megadni a parancssorban a kapcsoló után. Itt a következő állandókat használhatjuk:

G_OPTION_ARG_NONE A kapcsoló értéket nem kaphat, egyszerűen csak be- és kikapcsolni lehet.

G_OPTION_ARG_STRING A kapcsoló értéként karakterláncot kaphat.

G_OPTION_ARG_INT A kapcsoló értéként egész számot kaphat.

G_OPTION_ARG_CALLBACK A kapcsolóhoz egy függvény címét kell megadnunk, amely a kapcsoló szöveges értékét kiértékeli.

G_OPTION_ARG_FILENAME A kapcsolóhoz értéként állománynevet adhatunk meg.

G_OPTION_ARG_STRING_ARRAY A kapcsolóhoz értéként karakterláncokból álló listát adhatunk meg.

G_OPTION_ARG_FILENAME_ARRAY A kapcsoló értéke állománynevekből álló lista lehet.

arg_data A mező a kapcsolóhoz tartozó terület memóriabeli címét határozza meg. Ennek a címnek az értelmezése attól függ, hogy a **arg** mező milyen típust mutat:

G_OPTION_ARG_NONE Az **arg_data** mező értéke egy **gboolean** típusú változót jelöl a memóriában, ahol a G programkönyvtár jelzi, hogy ez a kapcsoló be volt -e kapcsolva vagy sem.

G_OPTION_ARG_STRING Az **arg_data** mező értéke egy **gchar *** típusú mutatót jelöl a memóriában, ahová a G programkönyvtár a kapcsolóhoz rendelt értéket elhelyezi.

G_OPTION_ARG_INT Az **arg_data** mező értéke egy **gint** típusú változót jelöl a memóriában, ahová a G programkönyvtár a kapcsolóhoz rendelt szám jellegű értéket elhelyezi.

G_OPTION_ARG_CALLBACK Az **arg_data** mező értéke egy **gboolean (*GOptionArgFunc)(const gchar *kapcsolónév, const gchar *érték, gpointer csoport, GError **hiba);** típusú függvényt jelöl a memóriában, amelyet a G programkönyvtár meghív, ha a felhasználó az adott kapcsolót használta a program indításakor.

G_OPTION_ARG_FILENAME Az **arg_data** mező értéke egy **gchar *** típusú mutatót jelöl a memóriában, ahová a G programkönyvtár a kapcsolóhoz rendelt állománynevet elhelyezi.

G_OPTION_ARG_STRING_ARRAY Az **arg_data** értéke egy **gchar **** típusú mutatót jelöl a memóriában, ahová a G programkönyvtár a karakterláncok címeit tartalmazó táblázat címét elhelyezi.

G_OPTION_ARG_FILENAME_ARRAY Az **arg_data** értéke egy **gchar **** típusú mutatót jelöl a memóriában, ahová a G programkönyvtár az állománynevek címeit tartalmazó táblázat címét elhelyezi.

120

description A kapcsolóhoz tartozó magyarázószöveget jelöli a memóriában, amelyet a G programkönyvtár a `--help` kapcsoló hatására kiír a kapcsoló neve mellé.

arg_description A kapcsoló értékének jele, amelyet a G programkönyvtár a `--help` kapcsoló hatására megjelenített sűgőban használ.

A bemutatott struktúra segítségével leírhatjuk a programunk kapcsolóinak tulajdonságait, a következő függvényekkel pedig kezelhetjük a program indításakor a felhasználó által megadott kapcsolókat:

GOptionContext *g_option_context_new(const gchar *szöveg); A függvény segítségével új környezetet hozhatunk létre a parancssori argumentumok kezelésére. Minden program számára legalább egy ilyen környezetet létre kell hoznunk.

A függvény argumentumaként átadott karakterlánc a `--help` hatására megjelenő sűgőszöveg elején jelenik meg (lásd a 19. példát).

void g_option_context_free(GOptionContext *környezet); A függvény segítségével a parancssori argumentumok kezelésére létrehozott környezetet semmisíthetjük meg, ha már nincs rá szükségünk. A függvénynek átadott argumentum az a mutató, amelyet a `g_option_context_new()` függvény adott vissza.

gboolean g_option_context_parse(GOptionContext *környezet, gint *argc, gchar *argv, GError **hiba);**

A függvény a programnak átadott parancssori kapcsolók és argumentumok értelmezését végzi el. A függvény argumentumai a következők:

környezet A függvény első paramétere a környezet mutatója, amelyet a `g_option_context_new()` függvénnyel hoztunk létre.

argc A függvény második paramétere a `main()` függvény – hagyományosan `argc` névvel létrehozott – első argumentumát – amely a program argumentumainak számát adja meg – jelöli a memóriában. A `g_option_context_parse()` függvény a program argumentumainak számát megváltoztatja, mert eltávolítja a már értelmezett kapcsolókat.

argv A függvény harmadik paramétere a `main()` függvény – hagyományosan `argv` nevű – második argumentumának címe.

hiba A függvény utolsó paramétere egy `GError` típusú mutató, ahova a függvény a hiba leírását helyezi el, ha a program paramétereinek értelmezése nem sikeres.

A függvény visszatérési értéke megadja, hogy a program paramétereinek értelmezése az adott környezet szerint sikeres volt -e. Ha a függvény visszatérési értéke **TRUE** a felhasználó által begépelt kapcsolók megfeleltek a környezetnek, ha **FALSE** a kapcsolók értelmezése során a függvény hibát talált.

A függvény az értelmezett kapcsolókat eltávolítja a program argumentumai közül, ezért a megmaradt argumentumokat – például a kapcsolók után található állományneveket – könnyedén tudjuk kezelni.

Ha a függvény hibát talált az argumentumok értelmezése közben, a hiba leírását tartalmazó adatszerkezet címét elhelyezi az utolsó argumentum által jelölt memóriacímre (lásd a 19. példa programját).

void g_option_context_set_ignore_unknown_options(GOptionContext *környezet, gboolean mellöz); A függvény segítségével beállíthatjuk, hogy az adott környezetben figyelmen kívül kívánjuk -e hagyni az ismeretlen kapcsolókat.

void g_option_context_add_main_entries(GOptionContext *környezet, const GOptionEntry *kapcsolók, const gchar *név); A függvény segítségével a környezethez adhatjuk a használni kívánt kapcsolók leírását. A függvény argumentumai a következők:

környezet Az első argumentum a környezetet jelöli a memóriában, amelyhez a kapcsolók listáját hozzá kívánjuk adni.

kapcsolók A kapcsolók listáját tartalmazó tömböt kijelölő mutató. Ezek a kapcsolók kerülnek be a környezetbe a függvény futása során.

A kapcsolókat tartalmazó tömb utolsó elemének **NULL** értékűnek kell lennie.

név A függvény utolsó paramétere megadja, hogy a magyarázószövegek fordításához milyen adatbázisra, milyen programnévre kell hivatkozni. Ez a karakterlánc a legtöbb esetben megegyezik az alkalmazás nevével.

19. példa. A következő példa bemutatja hogyan fogadhatjuk a parancs-sorban megadott argumentumokat a G programkönyvtár által biztosított eszközök segítségével. A bemutatott programrészlet futtatását mutatják be a következő sorok:

```
$ src/hbasic --help
Usage:
  hbasic [OPTION...] [FILE]
```

122

Basic language interpreter and compiler with debugging options.

Help Options:

-?, --help Show help options

Application Options:

-y, --debug-parser Debug parser.
-d, --dump Dump program and exit.
-O, --optimization=level Optimization level.

\$

Figyeljük meg, hogy a kapcsolók magyarázatát a szokásos formában kiírta a program. Ez a szolgáltatás automatikus, a G programkönyvtár függvényei biztosítják.

Figyeljük most meg, hogy mi történik, ha olyan kapcsolót használunk, amelyre a program nincs felkészülve:

\$ **src/hbasic -n**

```
** (hbasic:7644): WARNING **: Error parsing command line options:
Unknown option -n
```

\$

Amint látjuk a helytelen kapcsoló hatására a program futása megszakadt. A helytelen kapcsoló érzékelését elvégzi a G programkönyvtár, de azt, hogy mi történjék ilyen esetben, azt magunknak kell eldöntenünk.

Vizsgáljuk most meg a programrészletet, amely a parancssori kapcsolók elemzését végzik!

```
1  gboolean debug = FALSE;
2  gboolean dump = FALSE;
3  gint optm = 0;
4
5  int main(int argc, char *argv[]){
6      GError *error = NULL;
7      GOptionContext *context;
8      static GOptionEntry entries[] =
9      {
10         {
11             "debug-parser", 'y',
12             0,
13             G_OPTION_ARG_NONE, &debug,
14             "Debug parser.", NULL
15         },
16         {
```

```

17         "dump", 'd',
18         0,
19         G_OPTION_ARG_NONE, &dump,
20         "Dump program and exit.", NULL
21     },
22     {
23         "optimization", 'O',
24         0,
25         G_OPTION_ARG_INT, &optm,
26         "Optimization level.", "level"
27     },
28     {NULL}
29 };
30
31 context = g_option_context_new ("[FILE]\n\n"
32     "Basic language interpreter and compiler "
33     "with debugging options.");
34
35 g_option_context_add_main_entries(context,
36     entries,
37     PACKAGE);
38 if (!g_option_context_parse(context, &argc, &argv,
39     &error)){
40     g_warning("Error parsing command line options: %s",
41         error->message);
42     exit(EXIT_FAILURE);
43 }
44
45 g_option_context_free(context);
46
47 return yyparse();
48 }
```

A programrészlet 8–29. sorában láthatjuk a program által fogadott kapcsolók tulajdonságait leíró tömböt. Figyeljük meg a 28. sorban a tömb utolsó elemét jelölő `NULL` értéket, amely igen fontos szerepet tölt be. A tömb 13., 19. és 25. sorban figyelhetjük meg a hivatkozásokat azokra a változókra, amelyekben a G programkönyvtár a kapcsolókhoz rendelt értékeket elhelyezi.

A programban megfigyelhetjük hogyan hozzuk létre a kapcsolók értelmezésére szolgáló környezetet (31–33. sor), hogyan helyezzük el benne a kapcsolók leírását (35–37. sor), hogyan értelmezzük a kapcsolókat és hogyan vizsgáljuk meg, hogy az értelmezés talált-e hibát (38–43. sor) és hogyan szabadítjuk fel a lefoglalt erőforrásokat (45. sor).

Előfordulhat, hogy a Glade programmal készített, grafikus felülettel rendelkező program számára szeretnénk újabb parancssori kapcsolókat létrehozni. Ekkor a következő függvényeket szintén érdemes megismernünk:

`void gtk_init(int *argc, char ***argv);` E függvény a GTK+ programkönyvtár kezdeti beállítására szolgál, amelyet a GTK+ programkönyvtár szolgáltatásainak igénybe vétele előtt mindenképpen meg kell hívni (azonban nem feltétlenül ezzel a függvénnyel).

Ha a grafikus felülethez nem lehet kapcsolódni, akkor a `gtk_init()` megszakítja a program futását. Ha ez a viselkedés nem felel meg, a GTK+ kezdeti beállítását más függvény hívásával kell elvégeznünk.

A `gtk_init()` argumentumai a szabványos `main()` függvény argumentumainak címe. A függvény a mutatókon keresztül megkeresi az argumentumokat és az általa feldolgozott kapcsolókat és szöveges argumentumokat eltávolítja. Ez a roppant intelligens viselkedés lehetővé teszi, hogy a `gtk_main()` visszatérése után már ne kelljen foglalkoznunk azokkal az argumentumokkal, amelyeket a GTK+ programkönyvtár fogad.

A `gtk_init()` függvény hívását a Glade program elhelyezi a `main.c` állományban, de ez bizonyos esetekben egészen biztosan nem felel meg a céljainknak, ezért el kell távolítanunk és más függvénnyel kell helyettesítenünk.

`gboolean gtk_init_check(int *argc, char ***argv);` A függvény működése megegyezik a `gtk_init()` függvény működésével egy apró különbségtől eltekintve. Ez a függvény nem szakítja meg a program futását, ha nem lehet kapcsolódni a grafikus felhasználói felülethez, egyszerűen csak `FALSE` értéket ad vissza.

Nyilvánvaló, hogy ha a programunkat karakteres felületen is használhatóvá akarjuk tenni, akkor a `gtk_main()` függvény hívását le kell cserélnünk a `gtk_init_check()` függvény hívására és ha `FALSE` értéket kapunk vissza, akkor gondoskodnunk kell a karakteres felhasználói felület elindításáról.

`gboolean gtk_init_with_args(int *argc, char ***argv, char *paraméter_leírás, GOptionEntry *kapcsolók, char *nyelv, GError **hiba);` Ez a függvény szintén a GTK+ programkönyvtár kezdeti beállítását végzi el. Akkor használjuk, ha a programunk számára saját kapcsolókat szeretnénk létrehozni. A függvény argumentumainak és visszatérési értékének jelentése a következő:

visszatérési érték Igaz, ha a grafikus felülethez sikerült csatlakozni, hamis, ha nem.

argc A `main()` függvény első paraméterének címe.

argv A `main()` függvény második paraméterének címe.

paraméter_leírás A karakterlánc, amelyet a program a `--help` kapcsoló hatására megjelenő súgószövegben a program neve után kiír. Ebben a karakterláncban általában a kapcsolókon kívüli paramétereket – például bemenő állomány – és a program funkciójának egysoros leírását szokás megadni.

kapcsolók A program általunk készített kapcsolóinak leírását megadó tömb. A `gtk_init_with_args()` függvényt általában azért használjuk, mert saját parancssori kapcsolókat szeretnénk létrehozni, amelyet itt adhatunk meg.

nyelv A nyelv szöveges kódja, amelyet a `--help` kapcsoló hatására megjelenő súgószöveg lefordításakor szeretnénk megadni. Ennek a kapcsolónak a helyén általában `NULL` értéket adunk meg az alapértelmezett nyelv kiválasztására.

hiba Annak a mutatónak a címe, ahová a függvény az esetleges hiba meghatározását elhelyezi vagy `NULL`, ha nem kérjük ezt a szolgáltatást.

A következő példa bemutatja hogyan alakíthatjuk át a Glade által készített programot úgy, hogy az általunk létrehozott kapcsolókat és paramétereket is fogadja a GTK+ programkönyvtár kapcsolói mellett.

20. példa. A következő programrészlet a Glade által készített `main.c` állomány módosított változatát mutatja be. A részlet tartalmazza a program saját kapcsolóinak feldolgozását végző függvényt és azoknak a függvényeknek a hívását, amelyek a grafikus felhasználói felület kezdeti beállítására szolgálnak.

```

1  static void
2  program_init (int *argc, char ***argv)
3  {
4      gboolean hasX;
5      GError *error = NULL;
6      static GOptionEntry entries[] = {
7          {
8              "height", 'h',
9              0, G_OPTION_ARG_INT, &option_height,
10             "The height of the created image.", "PIXELS"
11          },
12          {

```

126

```

13     "width", 'w',
14     0, G_OPTION_ARG_INT, &option_width,
15     "The width of the created image.", "PIXELS"
16 },
17 {
18     "output", 'o',
19     0, G_OPTION_ARG_STRING, &option_outfilename,
20     "The output file name.", "FILENAME"
21 },
22 {
23     NULL
24 }
25 };
26
27 hasX = gtk_init_with_args(argc, argv,
28     "filename - convert binary to printable image",
29     entries,
30     NULL,
31     &error);
32
33 if (!hasX)
34     g_error("Could not connect to the screen.");
35
36 if (error != NULL)
37     g_error("Init error: %s", error->message);
38
39 if (*argc < 2)
40     g_error("Missing input file.");
41 else
42     option_infilename = (*argv)[1];
43 }
44
45 int
46 main (int argc, char *argv[])
47 {
48     GtkWidget *window1;
49
50 #ifdef ENABLE_NLS
51     bindtextdomain (GETTEXT_PACKAGE, PACKAGE_LOCALE_DIR);
52     bind_textdomain_codeset (GETTEXT_PACKAGE, "UTF-8");
53     textdomain (GETTEXT_PACKAGE);
54 #endif
55
56     /*

```



```

57      * Handling options.
58      */
59      program_init (&argc, &argv);
60
61      /*
62      * The following code was added by Glade to create one
63      * of each component (except popup menus), just so
64      * that you see something after building the project.
65      * Delete any components that you don't want shown
66      * initially.
67      */
68      window1 = create_window1();
69      gtk_widget_show(window1);
70      gtk_main ();
71      return 0;
72  }

```

A példa 1–41. soraiban található a `init_program()` függvény, ami a program és a GTK+ programkönyvtár kezdeti beállítását elvégzi. A függvény a `main()` argumentumainak címét kapja meg, hogy a feldolgozott kapcsolókat elvátolíthassa.

A 6–25. sorok közt a program számára bevezetett kapcsolók leírását láthatjuk. Figyeljük meg, hogy a programnak három újabb kapcsolót hozunk létre, amelyek egész és karakterlánc típusú argumentumokat is fogadnak. A 9., 14. és 19. sorok szerint ezeknek a kapcsoló-argumentumoknak a tárolására globális változókat használunk. (A kapcsolók kezelésére a globális változók tökéletesen alkalmasak, hiszen nyilvánvalóan csak egy példányban léteznek.)

A GTK+ programkönyvtár kezdeti értékének beállítására a 27–31. sorban hívott `gtk_init_with_args()` függvényt használjuk. A függvénynek átadott karakterlánc (28. sor) tanúsága szerint a programunk mindenképpen kell, hogy kapjon egy állománynevet, amihez nem tartozik kapcsoló. Ez a bemenő állomány neve.

A kezdeti beállítás után a 33–42. sorokban megvizsgáljuk az eredményt. Ha a `gtk_init_with_args()` visszatérési értéke hamis – azaz nem sikerült kapcsolódni a grafikus felülethez – a 34. sorban hibaüzenetet írunk ki, majd kilépünk. (A `g_error()` makró megszakítja a program futását.)

Ha a kezdeti beállítás nem sikerült – mert például a felhasználó hibás formában adta meg a kapcsolókat – a `gtk_init_with_args()` a hiba jellegét leíró adatszerkezet címét elhelyezi a `GError` típusú struktúrában, aminek a címét a 36. sorban vizsgáljuk. Hiba esetén a 37. sorban kiírjuk a hiba szöveges leírását, majd kilépünk. (A `GError` struktúra `message` mezője a szöveges leírást jelölő mutató.)

A `gtk_init_with_args()` az összes argumentumot eltávolítja, miután feldolgozta, ha tehát a felhasználó a program neve után nem adott meg bemenő állománynevet, akkor az `argc` változó értéke 1 lesz. A 39–42. sorokban megvizsgáljuk, hogy maradt-e argumentum. Ha igen, akkor azt tároljuk egy globális változóban (42. sor), ha nem, hibaüzenetet írunk ki és kilépünk (40. sor).

A `main()` függvényt a 45–72. sorokban láthatjuk. Ez a Glade által elhelyezett sorokat éppen úgy tartalmazza, mint a módosításokat, amelyek ahhoz szükségesek, hogy a program a saját kapcsolói kezelje.

Az 50–54. sorok közt a többnyelvű felhasználói felülettel rendelkező programok támogatásához szükséges függvényhívásokat tartalmazza, ahogyan azt a Glade elhelyezte az állományban.

A `program_init` függvény hívását az 59. sorban láthatjuk. Figyeljük meg, hogy a függvény argumentumként a `main()` argumentumainak címét kapja meg, hogy a feldolgozott kapcsolókat eltávolíthassa.

A 61–71. sorok közt a grafikus felhasználói felület megjelenítéséhez szükséges sorokat láthatjuk, ahogyan azokat a Glade elkészítette. A program ezen részéről a 7.1. szakaszban olvashatunk részletesebben.

A program parancssori kapcsolóiról szóló rész végén érdemes egy szót ejtenünk arról, hogy hogyan válasszuk meg a kapcsolókat, milyen rövid és hosszú neveket használjunk a jelölésükre. A legfontosabb alapelvnek, a „legkisebb meglepetés” elvének itt is érvényesülnie kell. A kapcsolókat úgy kell megválasztanunk, hogy a tapasztalt felhasználót ne érje meglepetés. A legjobb, ha olyan kapcsolókat, olyan neveket használunk, amelyekhez a felhasználónak más programokat használva volt alkalma hozzászokni.

5. fejezet

Osztályok készítése

A G programkönyvtárat használó programozó előbb utóbb eljut odáig, hogy saját osztályokat készítsen így teljes mértékben kihasználja az objektumorientált programozás adta előnyöket. A következő oldalakon arról olvashatunk, hogy milyen eszközökkel támogatja a G programkönyvtár az objektumorientált programozást.

Tudnunk kell azonban, hogy a GTK+ segítségével osztályok létrehozása nélkül is készíthetünk programokat. Az osztályok segítségével a programjainkat egyszerűbben, hatékonyabban elkészíthetjük – különösen ami a bonyolult, összetett alkalmazásokat illeti –, de nem okvetlenül kell használnunk ezeket az eszközöket. Ha az olvasó úgy érzi, hogy az itt bemutatott eszközök megismerése nehézségeket okoz nyugodtan átlépheti az egész fejezetet, az osztályok létrehozásának kérdését későbbre halasztva.

5.1. Egyszerű osztály készítése

Az osztályok létrehozását először egy egyszerű példán keresztül vizsgáljuk meg. A példa csak a legszükségesebb eszközöket használja és egy valóban egyszerű osztályt hoz létre. Nagyon fontos, hogy az itt bemutatott eszközöket pontosan ismerjük, hiszen a bonyolultabb osztályok, kifinomultabb eszközök használatához mindezek az ismeretek szükségesek.

A példa a `PipObject` nevű egyszerű osztályt két állományban, egy fejlécműveletben és egy C állományban hozza létre. A fejlécművelet tartalmazza mindazokat a definíciókat, amelyek az osztály használatához szükségesek, ezt az állományt tehát be kell majd töltenünk az `#include` utasítással valahányszor az osztályt használjuk.

130

```

1  #ifndef PIPOBJECT_H
2  #define PIPOBJECT_H
3
4  #ifdef HAVE_CONFIG_H
5  # include <config.h>
6  #endif
7
8  #include <glib-object.h>
9  #include <pip/pip.h>
10 #define DEBUG
11 #define WARNING
12 #include <pip/macros.h>
13
14 #define PIP_TYPE_OBJECT          \
15     (pip_object_get_type())
16
17 #define PIP_OBJECT(obj)          \
18     (G_TYPE_CHECK_INSTANCE_CAST((obj), \
19     PIP_TYPE_OBJECT, \
20     PipObject))
21
22 #define PIP_OBJECT_CLASS(klass)  \
23     (G_TYPE_CHECK_CLASS_CAST((klass), \
24     PIP_TYPE_OBJECT, \
25     PipObjectClass))
26
27 #define PIP_IS_OBJECT(obj)       \
28     (G_TYPE_CHECK_INSTANCE_TYPE((obj), \
29     PIP_TYPE_OBJECT))
30
31 #define PIP_IS_OBJECT_CLASS(klass) \
32     (G_TYPE_CHECK_CLASS_TYPE((klass), \
33     PIP_TYPE_OBJECT))
34
35 #define PIP_OBJECT_GET_CLASS(obj) \
36     (G_TYPE_INSTANCE_GET_CLASS((obj), \
37     PIP_TYPE_OBJECT, \
38     PipObjectClass))
39
40 typedef struct _PipObject      PipObject;
41 typedef struct _PipObjectClass PipObjectClass;
42
43 struct _PipObject

```

```

44  {
45      GObject      parent_instance;
46      gboolean     disposed;
47  };
48
49  struct _PipObjectClass
50  {
51      GObjectClass parent_instance;
52  };
53
54  /*****
55   * Public declarations.
56   *
57   *
58   *****/
59  GType pip_object_get_type(void);
60  GObject *pip_object_new (void);
61  #endif

```

Az állomány 1–2. soraiban a fejlőlmányt védő előfeldolgozó utasításokat olvashatjuk. Ez a szerkezet a szokásos módon védi az állományt a többszörös betöltéstől, a vége az állomány utolsó sorában olvasható.

Ezek után a 4–12. sorokban a szükséges fejlőlmányok betöltését láthatjuk. Be kell töltenünk mindazokat a fejlőlmányokat, amelyek a deklarációkat tartalmazzák a rendszeren található programkönyvtárakhoz és az alkalmazásunk egyéb részeihez. Ha osztályt készítünk, akkor mindeképpen szükségünk lesz a [glib-object.h](#) állományban található deklarációkra, így ezt az állományt valószínűleg be kell töltenünk.

Előfordulhat viszont, hogy magasabb szintű szolgáltatásokat használunk és így nem kell közvetlenül hivatkoznunk a [glib-object.h](#) állományra. Ha például betöltjük a [gtk/gtk.h](#) állományt, akkor már nincs szükség arra, hogy a [glib-object.h](#) állományt betöltsük, hiszen a GTK+ programkönyvtár a G programkönyvtár objektumrendszerére épül.

A példa 10–12. sorai az üzenetek kiírását segítő makrók deklarációit tartalmazzak, amelyeket az olvasónak megának kell elkészítenie.

Ezek után a 14. sortól kezdődően néhány igen fontos makró deklarációját olvashatjuk. E makró némelyikét az osztály ezen egyszerű, első változata nem használja, mégis érdemes őket már most elkészíteni, mert a későbbiekben igen hasznosnak fognak bizonyulni. A makrók a G és GTK+ programkönyvtáraknál megfigyelhető névsémát használják, hogy könnyebb legyen megjegyezni őket.

A 14–15. sorok a **PIP_TYPE_OBJECT** makróhoz hozzák létre. Ez a makró igen fontos, az általunk létrehozott osztályhoz tartozó típusazonosítót

adja vissza. A G programkönyvtár `GType` típusú, szám jellegű azonosítót rendel minden típushoz így minden osztályhoz is. Az általunk létrehozott osztály típusát a `PIP_TYPE_OBJECT` makró adja vissza, azaz valahányszor az általunk létrehozott osztály típusára akarunk hivatkozni ezt a makrót kell használnunk. Figyeljük meg azonban, hogy a `PIP_TYPE_OBJECT` makró csak a `pip_object_get_type()` függvény hívását írja elő, azaz a feladatot elnapoltuk, de nem oldottuk meg. A teljes megoldást a `pip_object_get_type()` függvény megvalósításával fogjuk elkészíteni.

A 17–20. sorokban a `PIP_OBJECT()` makró létrehozását láthatjuk. A makró a `G_TYPE_CHECK_INSTANCE_CAST()` makró segítségével végez típuskényszerítést az általunk létrehozott osztályra (azaz pontosabban az ilyen osztály objektumát jelölő mutatóra). Ez a makró szintén fontos, már láttuk milyen sokszor használjuk ezeket a típuskényszerítő makrókat.

A típuskényszerítéshez a 19. sorban felhasználjuk az imént bemutatott `PIP_TYPE_OBJECT` makrót, ami az általunk létrehozott típus típusazonosítóját adja vissza. Szintén felhasználjuk a típuskényszerítéshez a `PipObject` típusnevet, ami az osztályt jelöli (pontosabban az objektumok tárolására szolgáló adatszerkezet típusának neve). Ezt a típusnevet a későbbiekben fogjuk létrehozni.

A 22–25. sorokban szintén egy típuskényszerítő makrót hozunk létre, ez azonban nem az objektum típusár, hanem az osztály típusára változtatja az argumentum típusát. Ezt a makrót az osztály adatainak elérésekor használhatjuk amire a későbbiekben még visszatérünk.

A 27–29. sorokban létrehozott `PIP_IS_OBJECT()` makró megvizsgálja, hogy a paramétereként átadott mutató által jelölt objektum az általunk létrehozott osztályhoz tartozik-e. Függvények paramétereinek ellenőrzésekor, a polimorfizmus megvalósításakor nyilvánvaló hasznos lehet egy ilyen makró.

A típusellenőrzést a makró a 28. sorban található `G_TYPE_CHECK_INSTANCE_TYPE()` makró segítségével végzi. Már most tudnunk kell erről a makróról, hogy a G programkönyvtár által használt objektumot jelölő mutatót várja első paraméterként, egy olyan mutatót, ami egy `GObject` típusú adatszerkezetet jelöl a memóriában. Ha nem ilyen paramétert kap, a visszatérési értéke hibás lehet, sőt az is előfordulhat – ha az átadott memóriaterület mérete kisebb, mint a `GObject` adatszerkezet méreténél kisebb –, hogy a hívás miatt a programunk súlyos memóriahiba miatt leáll. Ha tehát a `PIP_IS_OBJECT()` makrónak tetszőleges objektumot jelölő mutatót adunk paraméterként semmi problémánk nem lehet, nem adhatunk viszont át akármilyen mutatót büntetlenül. E kérdésre az objektumok tárolásának bemutatása során még visszatérünk.

A 31–33. sorok közt a `PIP_IS_OBJECT_CLASS()` típusellenőrző makrót hozzuk létre. Ezzel a makróval az osztályt leíró adatszerkezet típusát

vizsgálhatjuk meg ellentétben a `PIP_IS_OBJECT()` makróval, ami az objektum – nem pedig az osztály – típusát vizsgálta.

A 35–38. sorokban létrehozott `PIP_OBJECT_GET_CLASS()` makró kifejtését láthatjuk. Ez a makró egy `PipObject` típusú objektumot jelölő mutatót kap paraméterül és visszaadja az objektum osztályát a memóriában tároló adatszerkezet címét. Nyilvánvaló, hogy akkor használhatjuk ezt a makró, ha az osztályadatokat vagy az osztály metódusait akarjuk elérni.

Már az eddig bemutatott makrókból is látható, hogy a G programkönyvtár az objektumot nyilvántartására két C nyelvű adatszerkezetet – két C struktúrát – használ. Az egyik adatszerkezet az objektumot, a másik adatszerkezet az osztályt írja le. Az osztály leírására használt adatszerkezetből mindig legfeljebb egy létezik, az objektumot leíró adatszerkezetből pedig mindig annyi, ahány objektumot létrehoztuk, azaz ahányszor példányosítottuk az osztályt (feltéve persze, hogy egyetlen objektumot sem semmisítettünk meg). Az objektumorientált programozásban jártas olvasók számára az osztályok tárolására szolgáló adatszerkezet kissé szokatlan lehet, hiszen ezeket az adatszerkezeteket az objektumorientált programozást támogató nyelvek általában elrejtik a programozó elől.

Nem kell azonban attól félnünk, hogy az osztály nyilvántartására szolgáló adatszerkezet használata túlságosan bonyolult lenne. A G programkönyvtár az osztály első példányosításakor (az első adott osztályhoz tartozó objektum létrehozásakor) automatikusan létrehozza az osztály tárolására szolgáló adatszerkezetet és valamikor az utolsó objektum megsemmisítése után fel is szabadítja azt. Mindez tehát automatikusan történik, a programozó pedig a megfelelő makró segítségével – ami a példánkban a `PIP_OBJECT_GET_CLASS()` makró – elérheti az osztályadatokat és osztálymetódusokat tároló adatszerkezetet.

Az osztály objektumának és az osztálynak a tárolására szolgáló adatszerkezetek létrehozását olvashatjuk a példa 40–52. soraiban. Amint láthatjuk a C nyelv struktúráit használjuk ilyen célból, ami elég nyilvánvaló, hiszen ezek a nyelvi elemek hasonlítanak a leginkább arra, amit az objektumorientált programozást támogató nyelvekben használunk.

Fontos kérdés azonban az öröklődés megvalósítása. A G programkönyvtár támogatja az öröklődést ami az objektumorientált programozás fontos alapeleme, ezért az osztály és az objektum tárolását is az öröklődés támogatásával kell elkészítenünk.

A példában megvalósított `PipObject` osztály a `GObject` osztály leszármazotta, azaz minden `PipObject` osztályba tartozó objektum egyben a `GObject` osztályba tartozó objektum is. (A `GObject` osztály a G programkönyvtár alaposztálya, minden osztály ennek az osztálynak a közvetlen vagy közvetett leszármazottja.) Az öröklődés ábrázolására a G programkönyvtár a C programozás egy bevett trükkjét használja, amennyiben az adatszerkezetek elején a szülő adatszerkezetet tartalmazza.

134

A példa 45. sorában láthatjuk, hogy a `PipObject` objektum memóriaterületének elején a `GObject` objektum adatszerkezete található. *Egy olyan mutató tehát, ami egy `PipObject` adatszerkezetet jelöl a memóriában egyben egy `GObject` objektum adatszerkezetet is jelöl.* Ilyen módon, a memóriamutatók segítségével megvalósítottuk az öröklődést (a `PipObject` adatszerkezet „örökölte” a `GObject` adatszerkezet mezőit) és a többalakúságot (a mutató, ami `PipObject` típust jelöle egyben `GObject` típust is jelöl). Ügyelnünk kell azonban, hogy a 45. és az 51. sorokban megfigyelhető sémát használjuk, azaz a szülő mezői a leszármazott mezői előtt, az adatszerkezetek elején megtalálhatók legyenek, ezeket a mezőket az egyes függvények a címük alapján érik el, ha valami mást tettünk ide, akkor minden reménytelenül összekeveredik.

A példánkban a `PipObject` a szülő objektumait csak egyetlen mezővel, a 46. sorban található `disposed` taggal bővíti, míg az osztályt egyáltalán nem bővíti, változatlanul hagyja.

A fejláomány végén két függvény deklarációját olvashatjuk. A `pip_object_get_type()` függvény a létrehozott osztály szám jellegű azonosítóját adja vissza, bár ahogyan azt láttuk ezt a függvényt a legtöbbször a `PIP_TYPE_OBJECT` makrón keresztül használjuk. A másik függvény a `pip_object_new()`, ami az osztály példányosítását végzi, azaz az osztályhoz tartozó objektumot hoz létre. Ez tulajdonképpen csak egy segédfüggvény, hiszen – ahogyan azt a későbbiekben bemutatjuk – a példányosítást a `g_object_new()` függvénnyel is elvégezhetjük.

A fejláomány után az osztályt megvalósító C állományt is bemutatjuk. A C program a fejláományok betöltését, egy globális változó deklarációját és néhány alapvetően fontos függvény megvalósítását tartalmazza.

```

1  #ifdef HAVE_CONFIG_H
2  #   include <config.h>
3  #endif
4
5  #include "PipObject.h"
6  #include <pip/pip.h>
7  #define DEBUG
8  #define WARNING
9  #include <pip/macros.h>
10
11 /*****
12  * Static declarations.
13  *
14  *
15  *****/
16 static GObjectClass *parent_class;
17 #define PIP_OBJECT_IS_DISPOSED(obj) ((obj)->disposed)

```

```

18 static void pip_object_class_init (PipObjectClass *);
19 static void pip_object_init      (PipObject *);
20 static void pip_object_dispose   (GObject *);
21 static void pip_object_finalize  (GObject *);
22
23 /**
24  * pip_object_get_type:
25  * @Returns: The registered type for #PipObject
26  *
27  * Registers #PipObject as a new type of the GObject
28  * system.
29  */
30 GType
31 pip_object_get_type(void)
32 {
33     static GType type = 0;
34
35     if (!type) {
36         PIP_DEBUG("Registering the 'PipObject' class.");
37         static const GTypeInfo info = {
38             sizeof(PipObjectClass),
39             (GBaseInitFunc) NULL,
40             (GBaseFinalizeFunc) NULL,
41             (GClassInitFunc) pip_object_class_init,
42             (GClassFinalizeFunc) NULL,
43             NULL,
44             sizeof (PipObject),
45             0,
46             (GInstanceInitFunc) pip_object_init,
47         };
48
49         type = g_type_register_static (
50             G_TYPE_OBJECT,
51             "PipObject",
52             &info, 0);
53     }
54
55     return type;
56 }
57
58 /**
59  * pip_object_class_init:
60  *
61  * Initializes the #PipObjectClass structure.

```

136

```

62  */
63  static void
64  pip_object_class_init(
65      GObjectClass *klass)
66  {
67      GObjectClass *gobject_class = (GObjectClass *) klass;
68
69      PIP_DEBUG("Initializing the class.");
70      parent_class = g_type_class_peek_parent(klass);
71      /*
72       * The GObject virtual functions.
73       */
74      gobject_class->dispose      = pip_object_dispose;
75      gobject_class->finalize     = pip_object_finalize;
76  }
77
78  /**
79   * pip_object_init:
80   *
81   * Instance initialization function.
82   */
83  static void
84  pip_object_init(
85      GObject *self)
86  {
87      PIP_DEBUG("Initializing object %p.", self);
88      /*
89       * Initializing tags.
90       */
91      self->disposed = FALSE;
92  }
93
94  /**
95   * pip_object_dispose:
96   *
97   * Drops all the references to other objects.
98   */
99  static void
100  pip_object_dispose(
101      GObject *object)
102  {
103      GObject *self = PIP_OBJECT(object);
104
105      PIP_DEBUG("Disposing object %p", self);

```

```

106  /*
107   * The dispose function can be called multiple times.
108   */
109  if (self->disposed)
110      return;
111  self->disposed = TRUE;
112  /*
113   * Drop references here.
114   */
115
116  PIP_DEBUG("Calling the parent's dispose function.");
117  G_OBJECT_CLASS(parent_class)->dispose(object);
118  }
119
120  /**
121   * pip_object_finalize:
122   *
123   * Instance destructor function.
124   */
125  static void
126  pip_object_finalize(
127      GObject *object)
128  {
129      PipObject *self = PIP_OBJECT(object);
130
131      PIP_DEBUG("Destroying object %p", self);
132      /*
133       * Free the resources here.
134       */
135
136      PIP_DEBUG("Calling the parent's finalize function.");
137      G_OBJECT_CLASS(parent_class)->finalize(object);
138  }
139
140  /*****
141   * Public functions.
142   *
143   *
144   *****/
145
146  /**
147   * pip_object_new:
148   * @returns: A new #PipObject
149   *

```

138

```

150  * Creates a new #PipObject.
151  */
152  GObject *
153  pip_object_new(void)
154  {
155      GObject      *retval;
156
157      PIP_DEBUG("Creating a new object.");
158      retval = g_object_new(
159          PIP_TYPE_OBJECT, NULL);
160      return retval;
161  }

```

Az állomány legfontosabb függvénye a 30–56. sorok közt olvasható `pip_object_get_type()` függvény, ami az új osztály nyilvántartásba vételét végzi és amelyet – amint azt a fejlécbeállításoknál már láttuk – általában a `PIP_TYPE_OBJECT()` makrón keresztül érünk el.

Ezt a függvényt igen sokszor hívjuk, a nyilvántartásba vételt azonban csak egyszer kell elvégezni, ezért a függvény egy statikus változóban (33. sor) tárolja a G programkönyvtár által adott típusazonosítót és egyszerűen visszaadja, ha a nyilvántartás már előzőleg megtörtént. Az első híváskor a 33. sor statikus változójának értéke 0, így a függvény az új osztályt a 37–52. sorok közt nyilvántartásba veszi. A nyilvántartásba vétel a `GTypeInfo` adatszerkezet létrehozásával és a `g_type_register_static()` függvény hívásával történik.

Egyszerű osztályról lévén szó a típusleíró `GTypeInfo` adatszerkezetben csak négy mezőt töltünk ki. A 38. és 44. sorokban az osztályt és az osztályhoz tartozó objektumot tartalmazó adatszerkezet méretét adjuk meg. Az osztályt és az objektumokat a legtöbb esetben dinamikus foglalt memóriaterületen hozzuk létre, ezért a G programkönyvtárnak tudnia kell mekkora helyet foglalnak ezek az adatszerkezet. Ez tehát egyszerű, eltekintve talán attól, hogy ha ezeket a mezőket rosszul adjuk meg elég misztikus és kiszámíthatatlan hibákat okozhatunk, ezért érdemes figyelmesen ellenőrizni a méreteket.

A másik két mező a 41. és 46. sorokban olvasható függvénymutatókat kapja értékül. A 41. sorban az osztály adatszerkezetét – az osztályadatok és osztálymetódusokat nyilvántartó adatszerkezetet – alapbeállítását végző függvény címét adjuk meg. Ezt a függvényt a G programkönyvtár az első objektum létrehozása előtt hívja az osztály előkészítésére. A második függvény a 46. sorban látható, az objektumok létrehozásakor, az objektum adatszerkezetének alapbeállítását végzi el. Ezt a függvényt az objektumorientált programozási módszertan általában konstruktornak hívja.

A típus nyilvántartásba vételét a 49–52. sorokban a

`g_type_register_static()` függvény végzi. A függvénynek első paraméterként a szülőosztály típusát adjuk át, amiből természetesen azonnal következik, hogy a szülőosztály nyilvántartásba vétele mindig a leszármazott osztály nyilvántartásba vétele előtt történik (a `g_type_register_static()` függvény hívása előtt a paraméterek értékét ki kell számolni). A függvény második paramétereként az új osztály nevét adjuk meg, harmadik paramétereként pedig az előzőleg létrehozott típusleíró adatszerkezet címét. Az utolsó paraméter csak azt határozza meg, hogy absztrakt osztályról van-e szó, ezért egyelőre megelégedhetünk azzal, hogy itt 0 értéket adunk át.

A függvény az 55. sorban visszaadja az osztály szám jellegű azonosítóját, amelyet a G programkönyvtár az osztály nyilvántartásba vétele során az osztályhoz rendelt.

Az állomány második legfontosabb függvénye az osztály alapbeállítását elvégző `pip_object_class_init()` függvény, amelyet a 64–76. sorok közt olvashatunk. Itt szintén egy igen egyszerű, a legfontosabb feladatokra szorítkozó megvalósítást olvashatunk.

A függvény paraméterül az általunk létrehozott osztályt leíró adatszerkezet címét kapja paraméterül. A paraméter neve `klass`, amit a G programkönyvtárat használó programozók előszeretettel használnak a `class` kulcsszó helyett, mert a `class` a C++ nyelv foglalt kulcsszava így a változónévként való használata lehetetlenné tenné, hogy a programot a C++ fordítóval is lefordítsuk. (A legtöbb esetben persze nincs szükségünk arra, hogy a programunkat C++ fordítóval fordítsuk le, de erre is érdemes gondolnunk amikor a programot elkészítjük.)

A függvény a 67. és a 70. sorokban két mutatót használ két osztály adatszerkezetének jelölésére. Nagyon fontos, hogy ezt a két mutatót és a feladatukat pontosab megértsük és ráadásul nem is teljesen nyilvánvaló a szerepük, ezért érdemes egy kissé részletesebben foglalkoznunk velük.

A 67. sorban létrehozott és inicializált `gobject_class` mutató az általunk létrehozott osztály – azaz a `PipObject` osztály – leíróját jelöli a memóriában de típusát tekintve megegyezik a szülőosztály adatszerkezetének típusával. *Ezzel a mutatóval az osztályunknak a szülőosztálytól örökölt osztályadatait és osztálymetódusait érhetjük el.* Erre nyilvánvalóan a dinamikus kötés és a virtuális függvények megvalósítása során van szükségünk.

A másik mutató (70. sor) a szülőosztály adatszerkezetét jelöli, típusa szintén a szülőosztály adatszerkezetének megfelelő. A 67. és a 70. sorban található mutatók típusa tehát megegyezik – ami könnyen félreérthetővé teszi a szerepüket –, a 67. sor mutatója azonban a mi osztályunkat, míg a 70. sor mutatója a a szülő osztály adatszerkezetét jelöli, ami igen nagy különbség.

A 70. sorban a szülő osztály adatszerkezetét jelölő mutatót egy statikus változóba tároljuk későbbi felhasználásra. Így az állomány függvé-

nyeiben bármikor hivatkozhatunk a szülő metódusaira, ami még ennél az egyszerű példánál is igen hasznos.

A `pip_object_class_init()` függvény két fontos sora a virtuális függvények felülírását végző két értékadás a 74–75. sorokban. A `GObject` osztály többek közt két virtuális – a leszármazott osztályban felülírható – függvényt hoz létre `dispose()` és `finalize()` néven az objektum megsemmisítésének automatizálására. Ezeket a függvényeket a `GObject` hozza létre és hívja, a hívás során azonban a függvények címét a `GObject` osztály osztályleíró adatszerkezetéből veszi a függvények címét – ahova a `GObject` osztálykonstruktor nyilvánvalóan elhelyezte azokat. Ez a módszer lehetővé teszi, hogy a leszármazott függvények címét felülírja, azaz virtuális függvényként felülbírálja a függvény megvalósítását.

A 74–75. sorokban az általunk létrehozott `PipObject` osztály örökölt `dispose()` és `finalize()` függvényeit írjuk felül a saját függvények-címének megadásával. Figyeljük meg, hogy természetesen nem a szülő osztály adatszerkezetét módosítjuk – az katasztrófához vezetne – hanem a saját osztályunk adatszerkezetét, azt `GObject` osztályleíróként kezelve a `PipObjectClass` adatszerkezet elején található `GObjectClass` típusú terület módosításával.

Igen fontos az is, hogy mit is csinálnak a `dispose()` és a `finalize()` függvények. Ezek a G programkönyvtár kétfázisú objektummegsemmisítő rendszerének részét képezik, azaz destruktorként használhatjuk őket.

A legtöbb objektumorientált nyelvben a destruktorkok rendszere egyszerű, az objektum megsemmisítése előtt a destruktork automatikusan meghívódik, hogy a programozó felszabadíthassa a konstruktorban foglalt erőforrásokat. A G programkönyvtár esetében a nagyobb rugalmasság érdekében a megsemmisítés folyamata kétfázisú. Először az objektum osztályához tartozó `dispose()` függvény hívódik meg, utána pedig az objektum osztályához tartozó `finalize()` függvény (ez után persze a G programkönyvtár felszabadítja az objektum tárolására használt memóriaterületet). A szokásos eljárás az, hogy a `dispose()` függvényben a `g_object_unref()` függvénnyel felszabadítjuk az objektum által foglalt objektumokat, a `finalize()` függvényben pedig felszabadítjuk az egyéb erőforrásokat. Ilyen módon az objektumok referenciaszámlálói közt megfigyelhető körköröség nem okoz problémát a megsemmisítésükkör.

Tudnunk kell viszont, hogy a G programkönyvtár nem garantálja, hogy a `dispose()` függvényt csak egyszer hívja a megsemmisítés során, ezért e függvénynek el kell viselnie, hogy az objektum megsemmisítése során többször is meghívódik. A szokásos eljárás az, hogy egy logikai változót helyezünk el az objektumban, ami tárolni fogja, hogy a `dispose()` függvény lefutott-e. Szokás az is, hogy a kívülről hívható függvények elején ellenőrizzük ennek a változónak az értékét, hogy érzékelhessük a hívó egy olyan objektumot próbál meg használni, aminek a megsemmisítése már elkezdődött.

A példánkban az objektumban elhelyezett **disposed** tagot használjuk annak jelzésére, hogy a **dispose()** végrehajtása megkezdődött. A változó deklarációját a fejléc 46. sorában találhatjuk. Ezt a tagot az objektum kezdeti beállítása során a C állomány 91. sorában hamis értékre állítjuk, majd a **dispose()** függvény védelmére használjuk a 109–111. soraiban.

A **dispose()** és a **finalize()** függvényekben találhatunk egy érdekes és elterjedten használt módszert arra is, hogy a virtuális függvényből a szülő virtuális függvényét hívva a munka egy részét „átharítsuk”. A **finalize()** függvényben például a 116. sorban – miután elvégeztük az osztályra jellemző feladatokat – a szülő hasonló függvényét hívjuk. Ehhez az osztály kezdeti beállítása során beállított **parent_class** változót használjuk fel. Hasonló módszert használhatunk a **finalize()** függvényben, ahol az objektum megsemmisítése során a destruktorki futása után hívjuk a szülő destruktorki. Ez a viselkedés az objektumorientált programozást támogató nyelvekben – így például a C++ nyelvben is – automatikus, a destruktorki után a szülő destruktorki fut, itt azonban nekünk kell erről gondoskodnunk.

Érdekes még egy pillantást vetnünk a 152–161. sorok közt megvalósított függvényre, ami kényelmi szolgáltatásként egy objektumot hoz létre az általunk létrehozott **PipObject** osztály példányaként. A visszatérési érték típusa a szülő típusával megegyező, ami megkönnyíti a hívó feladattal vagy legalábbis segíti a rugalmas program készítését.

A létrehozott osztályt egyszerűen kipróbálhatjuk ha létrehozunk egy objektumot és azonnal meg is semmisítjük a következő sorok segítségével.

```

1      GObject  *object;
2
3      object = pip_object_new();
4      g_object_unref(object);

```

E sorok hatására az üzeneteket megjelenítő makrók a következő sorokat írják a szabványos kimenetre:

```

      pip_object_new(): Creating a new object.
      pip_object_get_type(): Registering the 'PipObject' class.
      pip_object_class_init(): Initializing the class.
      pip_object_init(): Initializing object 0x8071490.
      pip_object_dispose(): Disposing object 0x8071490
      pip_object_dispose(): Calling the parent's dispose function.
      pip_object_finalize(): Destroying object 0x8071490
      pip_object_finalize(): Calling the parent's finalize function.

```

A sorok végigkövetése segíthet az osztály és az objektum léterhozásának és megsemmisítésének megértésében. Elég egyszerű programról lévén szó a folyamat magától értetődő.

5.2. Tulajdonságok rendelése az osztályhoz

A G programkönyvtár komoly segítséget nyújt a tulajdonságok kezeléséhez, amit mindenképpen érdemes kihasználnunk.

Az osztály nyilvántartásba vételekor megadhatjuk, hogy milyen tulajdonságokkal kívánjuk felruházni az osztályt, az osztály használata során pedig beállíthatjuk és lekérdezhettjük az egyes objektumok tulajdonságait. A G programkönyvtár a tulajdonságok kezelése közben az öröklődési hierarchiát is figyelembe veszi, azaz megkeresi, hogy az adott objektumhoz tartozó osztály szülői közül melyik rendelkezik az adott nevű tulajdonsággal és a megfelelő osztály függvényei segítségével állítja be a tulajdonság értékét.

A tulajdonságok használatához két függvényt kell megvalósítanunk – egyet a tulajdonságok lekérdezésére, egyet a tulajdonságok beállítására – és virtuális függvényként nyilvántartásba vennünk. Ezek után már csak egyszerűen létrehozunk és nyilvántartásba vesszük az egyes tulajdonságokat. A tulajdonságok használatára a következő lista alapján készíthetjük fel az általunk létrehozott osztályt:

1. A tulajdonságokat valahol tárolnunk kell. A legegyszerűbb esetben minden tulajdonság számára létrehozunk egy-egy mezőt az objektumban, ahogy a következő sorok is bemutatják:

```

1 struct _PipObject
2 {
3     GObject    parent_instance;
4     gboolean   disposed;
5     gchar      *name;
6     gint       ival;
7 };
    
```

A példában két tulajdonság tárolására hoztunk létre egy-egy tagváltozót, egy karakterlánc és egy egész típusú tulajdonságot fogunk itt tárolni.

2. Ha tagváltozókat adtunk az osztályhoz, akkor gondoskodnunk kell arról, hogy ezek alapbeállítása megtörténjen az objektum kezdeti értékének beállításakor. Ezt mutatják be a következő sorok:

```

1  static void
2  pip_object_init(
3      PipObject      *self)
4  {
5      PIP_DEBUG("Initializing %p", self);
6      /*
7       * Initializing tags.
8       */
9      self->disposed = FALSE;
10     self->name      = NULL;
11     self->ival      = 0;
12 }

```

3. A G programkönyvtár minden tulajdonsághoz egy egész számot rendel azonosítóul. Ezekről az azonosítókról nekünk kell gondoskodnunk, amire a legcélravezetőbb módszer egy `enum` típus létrehozása a C állományban. (Azért a C állományban, hogy a típus kívülről ne legyen látható.)

```

1  typedef enum {
2      PropertyName = 1,
3      PropertyIval
4  } MyProperties;

```

A tulajdonságokhoz rendelt azonosítószámoknak 0-nál nagyobbaknak kell lenniük, ezért az `enum` típus első tagjának értékét 1-re állítottuk.

Az `enum` típusnak nevet is adtunk, ezt azonban nem fogjuk használni. Ha viszont elhagynánk a nevet, akkor a C fordító figyelmeztetne a név hiányára a fordítás során.

4. A következő lépés a tulajdonságok lekérdezésekor hívandó függvény elkészítése. Ezt mutatják be a következő sorok:

```

1  static void
2  pip_object_get_property(
3      GObject      *object,
4      guint        property_id,
5      GValue       *value,
6      GParamSpec   *pspec)
7  {
8      PipObject    *self = PIP_OBJECT(object);
9

```

144

```

10     switch (property_id) {
11         case PropertyName:
12             g_value_set_string(value, self->name);
13             break;
14
15         case PropertyIval:
16             g_value_set_int(value, self->ival);
17             break;
18
19         default:
20             G_OBJECT_WARN_INVALID_PROPERTY_ID(
21                 object, property_id, pspec);
22     }
23 }
```

Amint látható a függvény egyszerűen visszaadja a szám azonosítónak megfelelő tulajdonság értékét a **GValue** típusú argumentumban tárolva.

5. Hasonlóképpen készíthetjük el a tulajdonságok beállítására szolgáló függvényt is.

```

1     static void
2     pip_object_set_property(
3         GObject      *object,
4         guint         property_id,
5         const GValue *value,
6         GParamSpec   *pspec)
7     {
8         PipObject *self = (PipObject *) object;
9
10        switch (property_id) {
11            case PropertyName:
12                g_free(self->name);
13                self->name = g_value_dup_string(value);
14                break;
15
16            case PropertyIval:
17                self->ival = g_value_get_int(value);
18                break;
19
20            default:
21                G_OBJECT_WARN_INVALID_PROPERTY_ID(
22                    object, property_id, pspec);
```

```
23     }
24 }
```

Amint látjuk a függvény egyszerűen csak eltárolja a kapott értéket a megfelelő helyre, egyébként semmi mást nem tesz. A gyakorlatban természetesen ennél bonyolultabb megoldásokat is használunk, hiszen az objektum tulajdonságának megváltoztatása más feladatokkal is együtt járhat.

6. Utolsó lépésként az osztály kezdeti beállítását végző függvényben nyilvántartásba vesszük a két virtuális függvényt és létrehozunk a két tulajdonságot.

```
1  static void
2  pip_object_class_init(
3      GObjectClass *klass)
4  {
5      GObjectClass *gobject_class =
6          (GObjectClass *) klass;
7      GParamSpec *pspec;
8
9      pip_object_parent_class =
10         g_type_class_peek_parent(klass);
11     /*
12      * The GObject virtual functions.
13      */
14     gobject_class->dispose      = pip_object_dispose;
15     gobject_class->finalize     = pip_object_finalize;
16     gobject_class->set_property =
17         pip_object_set_property;
18     gobject_class->get_property =
19         pip_object_get_property;
20     /*
21      * Registering properties.
22      */
23     pspec = g_param_spec_string (
24         "name",
25         "Name",
26         "The name of the object.",
27         NULL,
28         G_PARAM_READWRITE);
29     g_object_class_install_property(
30         gobject_class,
31         PropertyName,
```

146

```

32         pspec);
33
34     pspec = g_param_spec_int (
35         "ival",
36         "Ival",
37         "Some integer value.",
38         0,
39         G_MAXINT,
40         0,
41         G_PARAM_READWRITE);
42     g_object_class_install_property(
43         gobject_class,
44         PropertyIval,
45         pspec);
46 }
```

A példa a 16–19. sorokban nyilvántartásba veszi a tulajdonságok beállítására és lekérdezésére szolgáló függvényeket, majd a 23–45 sorokban két tulajdonságot rendel az osztályhoz.

A bemutatott módon létrehozott tulajdonságokat beállíthatjuk a `g_object_new()` és a `g_object_set()` függvényekkel és lekérdezhetjük a `g_object_get()` függvénnyel.

```

1  object = pip_object_new();
2  g_object_set(object,
3      "name",    "the name",
4      "ival",    10,
5      NULL);
6
7  g_object_get(object,
8      "name",    &name,
9      "ival",    &ival,
10     NULL);
11  PIP_MESSAGE("name = %s", name);
12  PIP_MESSAGE("ival = %d", ival);
```

5.3. Tagok elrejtése

Az osztályok létrehozásának egyik célja az egységbe záras, az objektum bizonyos tagjainak elrejtése a külvilág elől. Az eddig bemutatott eszközök alapján létrehozott osztályok azonban nem teszik lehetővé az adattagok elrejtését, ezért ezt külön kell megvalósítanunk.

A G programkönyvtár a rejtett tagok létrehozására a `g_type_class_add_private()` függvényt, használatára pedig a `G_TYPE_INSTANCE_GET_PRIVATE()` karót biztosítja. A rejtett tagokat azonban egyszerűen, saját eszközökkel is megvalósíthatjuk. Talán a legegyszerűbb módszer az, ha az osztály egyik tagjaként létrehozunk egy mutatót, ahol az objektum létrehozása során egy foglalt memóriaterület címét tároljuk. Ez a foglalt memóriaterület egy olyan struktúrát fog jelölni, amit az osztályt megvalósító C állományban definiálunk, így elemei az osztály C állományán kívül ismeretlenek lesznek. E módszer használatát mutatja be részletesebben a következő felsorolás:

1. Hozzunk létre egy mutatót az osztály definiálásakor. A mutató jelöljön egy struktúrát, ami az összes rejtett tagot fogja hordozni:

```

1 struct _PipObject
2 {
3     GObject          parent_instance;
4     struct _PipObjectPrivate *priv;
5 };

```

Ezt a szerkezetet a C fordító probléma nélkül lefodítja – annak ellenére, hogy a `_PipObjectPrivate` struktúrát még nem definiáltuk – hiszen a mutató mérete nem függ a jelölt memóriaterület típusától. A fejállományt tehát a fordításkor bármely állományba betölthetjük, nincs szükségünk a rejtett tagokat hordozó struktúra definíciójára.

2. Az osztály C állományában definiáljuk létre a rejtett tagokat hordozó struktúrát, sőt az a legjobb, ha mindjárt egy típust is létrehozunk e célra:

```

1 typedef struct _PipObjectPrivate PipObjectPrivate;
2 struct _PipObjectPrivate {
3     gboolean disposed;
4 };

```

A példában a struktúrának csak egyetlen mezője van – az objektumban tehát csak egyetlen rejtett tag lesz –, de ezt később könnyen bővíthetjük.

3. Az objektum kezdeti beállítását végző függvényben foglaljunk helyet a rejtett tagok számára és végezzük el azok kezdeti értékének beállítását is:

```

1 static void
2 pip_object_init(PipObject *self)

```


148

```

3  {
4      PIP_DEBUG("Initializing %p", self);
5      self->priv = g_new(PipObjectPrivate, 1);
6      self->priv->disposed = FALSE;
7  }

```

4. Az objektum megsemmisítése során természetesen gondoskodnunk kell e terület felszabadításáról is, de ez nem okozhat nehézséget:

```

1  static void
2  pip_object_finalize(GObject *object)
3  {
4      PipObject *self = PIP_OBJECT(object);
5
6      g_free(self->priv);
7      self->priv = NULL;
8
9      G_OBJECT_CLASS(pip_object_parent_class)->
10     finalize(object);
11 }

```

5. Az osztály C állományában az objektumok mezői közül a rejtett mezőket is igen egyszerűen, a mutató követésével elérhetjük, a tagok elrejtését tehát hatékonyan végeztük el:

```

1  PipObject *self = PIP_OBJECT(object);
2
3  if (self->priv->disposed)
4      return;

```

A példa sorai a **disposed** rejtett tag értékére hivatkozik a **priv** nevű mutatón keresztül.

A rejtett tagok létrehozása során persze felmerülhet a kérdés, hogy miért van egyáltalán szükség arra, hogy egyes mezőket elzárjunk a külvilág elől. Erre a kérdésre mindenki az élettapasztalatának, vérmérseketének megfelelően adhat választ, van azonban egy megközelítés, amit érdemes megfontolnunk.

Gondoljunk úgy a rejtett tagok létrehozására mint egy nyelvi eszközre, ami lehetővé teszi annak jelzését, hogy az adott tagok kívülről való elérése nem szerencsés. Bárki átteheti a C állományba létrehozott tagokat a fejállományba a nyilvános tagok közé, mindenki számára nyilvánvaló azonban, hogy az eredeti szerzőnek nem ez volt az elképzelése.

6. fejezet

XML állományok kezelése

Az XML (*extensible markup language*, bővíthető jelölőnyelv) egyszerű, szabvány által szabályozott nyelv, ami kitűnően használható adatok szöveges tárolására. Az XML rugalmas nyelvi szerkezetet használ, ember által is olvasható formában írja le az adatszerkezeteket, szöveges tárolást ír elő, így akár egy egyszerű szövegszerkesztő programmal is módosítható. Nem csoda, hogy sok szabvány és alkalmazás épül az XML állományok használatára. Érdeemes megismerkednünk az XML állományok kezelésére használható néhány eszközzel, amelyek hatékony adattárolást tesznek lehetővé a programozó számára.

Az XML programkönyvtár ([libxml](#) vagy más néven [libxml2](#)) hatékony és szabványos eszköztárat biztosít XML állományok kezelésére. Maga a programkönyvtár nem épül a G programkönyvtárra, nem része a G és GTK+ programkönyvtárak családjának, ezért az adatszerkezetei, valamint a függvényeinek elnevezése idegenszerű az eddig bemutatott eszközökkel összehasonlítva. Ráadásul a használatához a megfelelő fejlécműveleteket be kell töltenünk és a programkönyvtár programhoz való szerkesztéséről is külön kell gondoskodnunk. A nehézségeket azonban könnyen leküzdhetjük, nem beszélve arról, hogy a munka mindenképpen megéri a fáradságot, hiszen néhány egyszerű függvényhívás segítségével nagyszerűen használható adattároló eszközhöz juthatunk.

Mielőtt azonban részletesen megismerkednénk az XML programkönyvtárral, érdemes néhány szót ejtenünk az XML állományok szerkezetéről.

6.1. Az XML állomány szerkezete

Az XML állomány alapjában véve szöveges, felépítése, nyelvtani szerkezete egyszerű, ezért érdemes néhány példa alapján megismernünk.

Az első példa egy néhány sort tartalmazó XML állományt mutat be:

150

```
<?xml version="1.0" encoding="UTF-8"?>
<doc>
  <style name="Normal">
    <font>Times</font>
    <color>black</color>
  </style>
  <paragraph/>
</doc>
```

Az állomány elején – az első sorban – a `<? ?>` jelek közt az `xml` kulcs-szót találjuk, valamint a `version` (változat) és a `encoding` (kódolás) tulajdonságot. Ez a sor jelzi, hogy az `1.0` szabványváltozatnak megfelelő XML szabványnak megfelelő állományról van szó, ahol a karakterek kódolásására az UTF-8 szabványváltozatot használjuk. Az XML szabvány szerint az állomány elején ezeket az adatokat meg kell adnunk, ezért a legtöbb XML állomány elején ezt a sort – vagy egy nagyon hasonló sort – olvashatjuk.

Az XML állományokban a `< >` jelek közé írt szavakat, úgynevezett tagokat (*tag*, cédula, címke) figyelhetünk meg. A `< >` jelek közé az adott tag nevét kell írunk, valamint minden taghoz zárótagot kell létrehoznunk. A zárótag neve megegyezik a tag névvel, azzal a különbséggel, hogy a zárótag névének a `/` jellel kell kezdődnie. A `<doc>` tag zárótagja például a `</doc>`, a `<font>` tag zárótagja pedig a `</font>` lesz.

A tagokból és a zárótagokból álló kifejezések egymásbaágyazhatók, az egymásbaágyazott részekből faszerkezet alakítható ki. A fa gyökere az XML dokumentum gyökértagja, amelyből minden XML dokumentum ki-zárólag egyet tartalmazhat. A következő sorok egy hibás XML állományt mutatnak be:

```
<?xml version="1.0" encoding="UTF-8"?>
<doc></doc>
<font></font>
```

A példa soraiban a 2. és a 3. sor is gyökérelem és mivel csak egyetlen gyökérelem lehet, az állomány hibás. A következő sorokban a hibát javítottuk:

```
<?xml version="1.0" encoding="UTF-8"?>
<font>
  <doc></doc>
</font>
```

A példa gyökéreleme a „font” nevű tag, annak leszármazottja pedig a „doc” nevű tag a 3. sorban.

Az XML állományba a tagok és zárótagok közé szöveget is írhatunk. A szöveg az adott tag szöveges értéke lesz. A rövidség érdekében azokat a

tagokat és zárótagokat, amelyek közt egyetlen karakter sincs, összevonjuk. Az összevont kezdő- és zárótag a tag nevét tartalmazza, ezt pedig a / jel követi. A `<paragraph/>` például a `<paragraph></paragraph>` összevont változata, azaz egy üres, „paragraph” nevű tagpár.

Minden tagnak tetszőleges jellemzője, attribútuma is lehet. Az attribútumokat a kezdő tag záró `>` karaktere elé, a tag neve után kell írunk, mégpedig úgy, hogy megadjuk az attribútum nevét, majd egy `=` jel után a `" "` vagy a `' '` jelek közé írt értékét. A következő sorok ezt mutatják be:

```
<?xml version="1.0" encoding="UTF-8"?>
<database type="dictionary" version="1.0">
  <title>Magyar Angol Szótár</title>
</database>
```

A példa 2. sorában megfigyelhetjük, hogy a „database” nevű tagnak két attribútuma van, egy „type” és egy „version” nevű. A „database” tagon belül található a „title” nevű leszármazottja, amelynek nincsen egyetlen attribútuma sem, csak szöveges értéke.

Az XML szabvány szerint a tagok neveiben betűk és számjegyek is állhatnak, a nevek azonban nem kezdődhetnek számjeggyel. A nevek betűinek esetében figyelembe kell vennünk a kis- és nagybetűk közti különbséget. A betűk használatát nem kell az angol nyelv betűire korlátoznunk, gyakorlatilag tetszőleges természetes nyelv karakterkészlete használható. (Természetesen a legtöbb esetben az angol nyelvet vesszük alapul, hogy a tagok nevei könnyen érthetővé tegyék az állomány szerkezetét.)

A tagok neveiben az elválasztásra az aláhúzás karaktert (`_`), az elválasztójelet (`-`) és a pontot (`.`) használhatjuk. Más írásjelet – például időzőjelet, szóközt – a tag neve nem tartalmazhat. A tagok neve ráadásul nem kezdődhet az `xml` karakterekkel – sem kis- sem nagybetűkkel, sem pedig azok kombinációival –, mert az ilyen nevek az XML szabványokat kezelő W3C számára vannak fenntartva.

A tagok atribútumainak neveire ugyanazok a szabályok vonatkoznak, mint a tagok neveire és a legtöbb esetben szintén valamilyen angol szót választunk, ami jól kifejezi, hogy mit ad meg az adott jellemző.

Nyilvánvalóan félreértésre adna lehetőséget, ha a tagok szöveges értékében, a kezdő és a zárótag közt a `<` karaktert tartalmazó szöveget írnanánk, vagy az attribútumok szöveges értékében a `"` jelet használnánk. Az ilyen problémák elkerülésére az XML szabvány karakterazonosítókat (*entity reference*, entitásazonosító) használ. A szabvány által meghatározott XML karakterazonosítókat és jelentésüket a 6.1. táblázatban mutatjuk be.

Rövidítés	Jelentés
<	Kisebb mint: <.
>	Nagyobb mint: >.
&	Ampersand: &.
"	Kettős idézőjel: "
'	Aposztróf: '.

6.1. táblázat. Az XML karakterazonosítók és a jelentésük

6.2. Az XML programkönyvtár egyszerű használata

Az XML programkönyvtár nem tartozik szorosan a G vagy a GTK programkönyvtárhoz, amit az eltérő függvénynevek és különálló fejláblományok is jeleznek, ezért a programkönyvtár használatáról az alkalmazás megírásakor külön kell gondoskodnunk.

6.2.1. A fordítás előtti beállítás

A programkönyvtár használatának első lépéseként gondoskodnunk kell arról, hogy a fordítás előtti beállításokat végző `configure` program az XML programkönyvtárat megkeresse, a programkönyvtár fejláblományainak és a lefordított programkönyvtárat tartalmazó állományoknak a helyét feljegyezze. Ehhez a következő sorokat helyezzük el a `configure.in` állományban:

```
pkg_modules="libxml-2.0"
PKG_CHECK_MODULES(PACKAGE, [$pkg_modules])
AC_SUBST(PACKAGE_CFLAGS)
AC_SUBST(PACKAGE_LIBS)
```

A `PKG_CHECK_MODULES()` makró hatására a `configure` program a `pkg-config` program segítségével megkeresi és feldolgozza az XML programkönyvtár legfontosabb adatait tartalmazó `libxml-2.0.pc` állományt. Az állományrészlet utolsó két sorában látható `AC_SUBST()` makrók hatására a C fordító a fordítás előtti beállítás során elkészített `Makefile` alapján megkapja az XML programkönyvtár használatához a fordítás és a szerkesztés során szükséges kapcsolókat.

A bemutatott módszerrel egy időben több programkönyvtár használatát is lehetővé tehetjük. A következő sorok például két programkönyvtárra vonatkoznak:

```
pkg_modules="gtk+-2.0 >= 2.0.0 libxml-2.0"
```

```
PKG_CHECK_MODULES(PACKAGE, [$pkg_modules])
AC_SUBST(PACKAGE_CFLAGS)
AC_SUBST(PACKAGE_LIBS)
```

Az első sor alapján a fordításhoz felhasználjuk a GTK programkönyvtárat (legalább 2.0.0 változattal) és az XML programkönyvtárat is. Ráadásul a `pkg-config` program a programkönyvtárak közti függőségeket is figyelembe veszi, így minden programkönyvtárat megkeres, amire a megadott programkönyvtárak használatához szükségünk van.

A `configure.in` állomány módosítása után futtatnunk kell az `autoconf` és az `automake` programokat, hogy az új beállításoknak megfelelő állományok elkészüljenek. Ha az alkalmazást a Glade segítségével hoztuk létre, akkor a projektkönyvtárban található `autogen.sh` program elvégzi helyettünk ezt a feladatot, elegendő tehát csak ezt futtatnunk.

Ha ezek után a projektkönyvtár `configure` nevű programját futtatjuk, az megkeresi az XML programkönyvtárat és gondoskodik a helyes `Makefile` állományok elkészítéséről.

6.2.2. A fejláományok betöltése

Az XML programkönyvtár által létrehozott típusok és függvények használatához természetesen be kell töltenünk a megfelelő fejláományokat. Maga a programkönyvtár igen sok eszközt hoz létre, az általunk bemutatott példák lefordításához azonban elegendő a következő sorokat elhelyezni a forrásprogram elején:

```
#include <libxml/xmlmemory.h>
#include <libxml/parser.h>
#include <libxml/xpathInternals.h>
#include <libxml/tree.h>
```

Ezek a fejláományok a legfontosabb függvényeket tartalmazzák, amelyek lehetővé teszik az XML állományok létrehozását, betöltését, valamint hatékony eszközöket adnak az XML állományban található adatok lekérdezésére és megváltoztatására is.

6.2.3. A dokumentum létrehozása és mentése

Az XML programkönyvtár segítségével többek közt összetett adatszerkezeteket hozhatunk létre a memóriában és azokat XML állományként el is menthetjük. A memóriában tárolt dokumentumok létrehozásához és az XML állományok mentéséhez a következő függvényeket érdemes ismer-nünk.

`int xmlKeepBlanksDefault(int val);` A függvény segítségével befolyásolhatjuk, hogy az XML programkönyvtár hogyan kezelje az elhagyható szöközőket és újsor karaktereket.

A függvény paramétere 0 vagy 1. Ha 0 értéket adunk, az XML programkönyvtár nem tartja meg az elhagyható karaktereket, ha 1 értéket, akkor viszont igen. A függvény visszatérési értéke megadja a függvény hívása előtti beállítást.

Nagyon fontos, hogy ennek a függvénynek a hívása befolyásolja az XML programkönyvtár által létrehozott XML állományok tördelését. Ha a függvényt 1 argumentummal hívjuk a létrehozott állományban a programkönyvtár nem használ automatikus beljebb írást (*autoindentation*) a szerkezet jelölésére. A legjobb, ha a munka megkezdése előtt mindig hívjuk ezt a függvényt mégpedig 0 paraméterrel.

A függvényt szintén érdemes 0 paraméterrel meghívni mielőtt XML állományt töltenénk be, hogy az XML programkönyvtárat felkészítsük a beljebb írással szerkesztett állomány kezelésére. Ha az indentálásra használt újsor karakterek és szöközők szöveges értéként megjelennek a dokumentumban, akkor valószínűleg elfelejtettük ezt a függvényt használni.

`xmlDocPtr xmlNewDoc(const xmlChar *változat);` A függvény segítségével új XML dokumentumot hozhatunk létre a memóriában.

A függvény argumentuma szöveges értéket jelöl a memóriában, ami megadja a használt XML szabvány változatának számát. Ha a paraméter értéke `NULL`, a függvény az "1.0" karakterláncot használjuk változatként.

A függvény visszatérési értéke a létrehozott dokumentumot jelöli a memóriában. Ezt az értéket meg kell tartanunk, mert ezzel tudjuk módosítani és menteni a dokumentumot.

`void xmlFreeDoc(xmlDocPtr doc);` A függvény segítségével megsemmisíthetjük a memóriában elhelyezett dokumentumot, felszabadítjuk a tárolására használt memóriaterületeket.

A függvény argumentuma a megsemmisítendő dokumentumot jelöli a memóriában.

`int xmlSaveFormatFileEnc(const char *fájlnév, xmlDocPtr doc, const char *kódolás, int formátum);` A függvény segítségével a memóriában található XML dokumentumot elmenthetjük állományba vagy kiírathatjuk a szabványos kimenetre. A mentésre az XML programkönyvtár többféle lehetőséget is biztosít, de a legtöbb esetben elegendő ennek az állománynak a használata.

A függvény első paramétere az állomány neve, amelybe a dokumentumot menteni akarjuk. A függvény az állományt automatikusan létrehozza, ha létezne, akkor pedig törli.

Ha a dokumentumot a szabványos kimenetre akarjuk kiíratni, akkor az állomány nevéként a "-" értéket kell megjelölnünk.

A függvény második paramétere a menteni kívánt dokumentumot jelöli a memóriában.

A dokumentum harmadik paramétere a dokumentum kódolását meghatározó karakterláncot jelöli a memóriában. Az XML programkönyvtár ezt a karakterláncot elhelyezi a dokumentum szövegében. A legtöbb esetben a harmadik paraméter helyén a "UTF-8" állandót használjuk.

A függvény negyedik paramétere megadhatjuk, hogy használni akarjuk-e az XML programkönyvtár automatikus szövegformázását. A legtöbb esetben itt az 1 értéket adjuk meg, hogy az XML programkönyvtár automatikusan formázza az XML állományt szóközzel és újsor karakterekkel.

```
xmlNodePtr xmlNewDocNode(xmlDocPtr doc, xmlNsPtr ns,
    const xmlChar *név, const xmlChar *tartalom);
```

A szabvány szerint minden XML dokumentumban pontosan egy gyökércsomópontnak vagy más néven gyökérelemnek kell lennie. A gyökércsomópontot ezzel a függvénnyel hozhatjuk létre.

A függvény első paramétere a dokumentumot jelöli a memóriában.

A függvény második paramétere az előkészített névteret jelöli a memóriában, amelyet a gyökérelemhez kívánunk rendelni. Egyszerűbb esetekben - ha nem akarunk névteret használni - a második paraméter helyén NULL értéket adunk meg. (A névterek használatára a későbbiekben még visszatérünk.)

A függvény harmadik paramétere a gyökércsomópont nevét jelöli a memóriában.

A függvény negyedik paramétere a gyökérelem tartalmát jelöli a memóriában. A legtöbb esetben a gyökérelemet tartalom nélkül hozzuk létre, így a negyedik paraméter értéke NULL.

A függvény visszatérési értéke a létrehozott csomópontot jelöli a memóriában. Ezt a mutatót el kell helyeznünk a dokumentumban az xmlDocPtr által jelölt memóriaterület children mezőben.

A következő példa bemutatja hogyan hozhatunk létre XML dokumentumot az XML programkönyvtár segítségével.

21. példa. A következő sorok létrehoznak és állományba mentenek egy egyszerű XML dokumentumot.

156

```

1 void doc_create(void)
2 {
3     xmlDocPtr document = NULL;
4
5     document = xmlNewDoc("1.0");
6     document->children = xmlNewDocNode(document, NULL,
7                                         "root", NULL);
8
9     if (!xmlSaveFormatFileEnc("tmp.xml", document,
10                              "UTF-8", 1)){
11         g_error("Error saving document: %m");
12     }
13 }

```

A függvény által létrehozott állomány tartalma a következő:

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <root/>

```

Amint látható az állomány tartalma nagyon egyszerű, viszont követi az XML szabványt, így könnyen feldolgozható és bővíthető.

6.2.4. A dokumentum betöltése

Az XML állományok szabványos nyelvtan alapján épülnek fel és így minden különösebb előkészítés nélkül automatikusan értelmezhetők. Az XML programkönyvtár természetesen tartalmazza a megfelelő függvényeket XML állományok értelmezésére, betöltésére. Ha XML állományt akarunk betölteni, használhatjuk a következő függvényt.

`xmlDocPtr xmlParseFile(const char *fájlnev);` A függvény segítségével memóriában elhelyezett dokumentumot hozhatunk be az állományban elhelyezett vagy a szabványos bemenetről olvasott sorok értelmezésével. A beolvasott dokumentumot a későbbiekben módosíthatjuk és kiírhatjuk.

A függvény paramétere a betöltendő állomány nevét jelöli a memóriában. Ha az állomány neve helyén a "-" nevet adjuk meg, a függvény az adatsorokat a szabványos bemenetről olvassa.

A függvény visszatérési értéke a függvény által létrehozott és a memóriában elhelyezett dokumentumot jelöli, ha az adatsorok beolvasása és értelmezése sikeres volt, `NULL` érték, ha a művelet közben hiba történt. Ha az állomány nem teljesíti az XML szabványt, a visszatérési érték `NULL`.

6.2.5. A dokumentum bővítése

Az XML programkönyvtár összetett, rugalmasan használható eszközöket biztosít az XML dokumentumban található adatok lekérdezésére és módosítására. Az eszközök közül az egyszerűség érdekében csak az okvetlenül szükséges és egyszerű függvényeket vesszük sorra.

A következő függvények segítségével új csomópontokat hozhatunk létre a dokumentumban és beállíthatjuk a létrehozott csomópontok tulajdonságait.

`xmlChar *xmlEncodeEntitiesReentrant(xmlDocPtr doc, const xmlChar *szöveg);` A függvény segítségével a dokumentumban elhelyezendő szövegben a tiltott karaktereket helyettesíthetjük a szabvány által előírt rövidítésükkel. A dokumentum lekérdezése során az eredeti szöveg természetesen visszaállítható.

A függvény első paramétere a dokumentumot, második paramétere pedig az átalakítandó szöveget jelöli a memóriában.

A függvény visszatérési értéke az átalakított szöveget jelöli a memóriában, ami dinamikusan foglalt memóriaterületen van. Ezt a memóriaterületet használat után felszabadíthatjuk az `xmlFree()` vagy a `free()` függvénnyel.

`void xmlFree(void *memória);` A függvény az XML programkönyvtár által lefoglalt dinamikus memóriaterületek felszabadítására szolgál, működésében tulajdonképpen megegyezik a szabványos programkönyvtár `free()` függvényével.

Ez a függvény egyszerűen csak felszabadítja a memóriaterületet, az összetett XML adatszerkezetek megsemmisítésére más függvényeket kell használnunk.

`xmlNodePtr xmlNewChild(xmlNodePtr szülő, xmlNsPtr névtér, const xmlChar *név, const xmlChar *tartalom);` A függvény segítségével új elemet, új csomópontot hozhatunk létre a dokumentumban.

A függvény első paramétere az új elem szülőelemét jelöli a memóriában. Az új elem ennek a csomópontnak a leszármazottjaként, ezen az elemen belül jön létre. Ha a szülőelemnek már vannak leszármazottjai, akkor az új csomópont az utolsó helyre kerül, a meglévő leszármazottak után utolsóként jelenik meg. Ez fontos lehet, hiszen az XML állományban sokszor a csomópontok sorrendje is számít.

A függvény második paramétere az új elem számára előzőleg létrehozott névteret jelöli a memóriában. Ha a második paraméter értéke `NULL`, az új csomópont a szülő névterét örökli, ezért a legtöbb

esetben egyszerűen `NULL` értéket adunk meg névtérként. A névterek használatára a későbbiekben még visszatérünk.

A függvény harmadik paramétere az új elem nevét határozza meg. A névnek a szabvány által meghatározott feltételeknek kell megfelelnie.

A függvény negyedik paramétere az új csomópont által hordozott szöveget jelöli a memóriában vagy `NULL`, ha az elemen belül nem akarunk szöveget elhelyezni.

A negyedik paraméter által kijelölt szövegnek a szabvány alapján nem szabad tiltott karaktereket tartalmaznia. Ha nem vagyunk biztosak benne, hogy a szöveg megfelel-e a szabványnak, a megfelelő átalakításokat el kell végeznünk a `xmlEncodeEntitiesReentrant()` függvény segítségével. Úgy tűnik azonban, hogy az XML programkönyvtár a dokumentációval ellentétben automatikusan elvégzi a tiltott karakterek cseréjét, ha ezt a függvényt használjuk.

A függvény visszatérési értéke a létrehozott új elemet jelöli a memóriában.

`xmlAttrPtr xmlSetProp(xmlNodePtr elem, const xmlChar *név, const xmlChar *szöveg);` A függvény segítségével új tulajdonságot rendelhetünk a dokumentum elemeihez és meghatározhatjuk a tulajdonság értékét.

A függvény első paramétere azt az elemet jelöli a memóriában, aminek a tulajdonságát be szeretnénk állítani.

A függvény második paramétere a tulajdonság nevét jelöli a memóriában. A névnek a szabvány szerint kell felépülnie, amit programkönyvtár nem ellenőriz.

Ha az első paraméter által jelölt elem már rendelkezik a második paraméter által meghatározott nevű tulajdonsággal, a függvény a tulajdonság értékét felülírja, ha nem, a függvény új tulajdonságot hoz létre.

A függvény harmadik paramétere a tulajdonság szöveges értékét jelöli a memóriában. Az `xmlSetProp()` a `xmlNewChild()` függvényhez hasonlóan automatikusan lecseréli a szövegben található tiltott karaktereket a megfelelő rövidítésre, bár ezt a szolgáltatást a dokumentáció nem részletezi.

A függvény visszatérési értéke olyan memóriacím, ami a tulajdonság tárolására szolgáló adatszerkezetet jelöli a memóriában. Ezt a memóriacímet általában egyszerűen eldobjuk.

`int xmlUnsetProp(xmlNodePtr elem, const xmlChar *név);` A függvény segítségével a csomópont adott néven bejegyzett tulajdonságát törölhetjük.

A függvény első paramétere a módosítandó elemet, a második paramétere pedig a törlendő tulajdonság nevét jelöli a memóriában.

A függvény visszatérési értéke 0, ha a művelet sikerült, -1 hiba esetén.

Az XML dokumentum egyszerű bővítésére szolgáló függvények használatát mutatja be a következő példa.

22. példa. A következő függvény néhány új csomópontot hoz létre a paraméterként kapott mutató által jelölt dokumentumban a gyökérelemből indulva.

```

1 void add_test(xmlDocPtr doc)
2 {
3     xmlChar      *tmp;
4     xmlNodePtr   node;
5
6     node = xmlNewChild(doc->children, NULL, "if", NULL);
7     xmlSetProp(node, "type", "numeric");
8
9     tmp = xmlEncodeEntitiesReentrant(doc, "8 < 9");
10    xmlNewChild(node, NULL, "condition", tmp);
11    xmlFree(tmp);
12
13    xmlNewChild(node, NULL, "true", "a");
14    xmlNewChild(node, NULL, "false", "b");
15 }
```

Ha a függvény olyan dokumentumot kap, amelyben csak a gyökérelem található meg, a következő dokumentumot kapjuk eredményül:

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <root>
3     <if type="numeric">
4         <condition>8 &lt; 9</condition>
5         <true>a</true>
6         <false>b</false>
7     </if>
8 </root>
```

A programrészlet működése jól követhető az XML állomány segítségével.

6.2.6. A bejárás és a lekérdezés

Az XML programkönyvtár az XML dokumentumot a memóriában összetett adatszerkezetben tárolja. A dokumentumot a programkönyvtár `xmlDocPtr` típusú mutatóval jelölt `xmlDoc` típusú adatszerkezetben tárolja. Az adatszerkezet felépítése nem igazából fontos, csak egyetlen eleme van, ami mindenképpen említésre érdemes, mégpedig a `xmlNode` típusú adatszerkezetet kijelölő `xmlNodePtr` típusú, `children` nevű mutató, ami a dokumentum gyökérelemét jelöli a memóriában.

A dokumentum nagyjából az `xmlNode` típusú adatszerkezetekben tárolt csomópontok halmazából áll, amelyek a bennük található mutatók segítségével faszerkezetbe vannak rendezve. Ha a dokumentumban keresni akarunk, be akarjuk járni, módosítani szeretnénk, akkor meg kell ismerkednünk a dokumentumelemek tárolására szolgáló `xmlNode` adatszerkezettel és azzal is, hogyan hivatkoznak az ilyen adatszerkezetek egymásra a bennük található `xmlNodePtr` típusú mutatók segítségével. Az `xmlNode` adatszerkezet legfontosabb elemei a következők:

A <code>xmlNode</code> struktúra	
<code>void *_private</code>	Szabadon használható.
<code>xmlElementType type</code>	A csomópont típusa.
<code>const xmlChar *name</code>	A csomópont neve.
<code>struct _xmlNode *children</code>	Az első leszármazott.
<code>struct _xmlNode *last</code>	Az utolsó leszármazott.
<code>struct _xmlNode *parent</code>	A szülő.
<code>struct _xmlNode *next</code>	A következő testvér.
<code>struct _xmlNode *prev</code>	Az előző testvér.
<code>struct _xmlDoc *doc</code>	A dokumentum.
<code>xmlNs *ns</code>	Névtér.
<code>xmlChar *content</code>	A szövegtartalom.
<code>struct _xmlAttr *properties</code>	A tulajdonságok.
<code>unsigned short line</code>	A sor száma a fájlban.

Az egyes részek pontos jelentését tárgyalja a következő lista:

_private Az alkalmazásprogramozó számára fenntartott mező, amelyet a programunkban arra használunk amire csak akarunk. Mindazokat az adatokat, amelyeket a csomópontokhoz kívánunk rendelni, el kell helyoznünk egy tetszőleges memóriaterületen – például egy struktúrában –, majd a memóriaterület címét egyszerűen el kell helyoznunk ebben a mezőben.

type Az `xmlNode` típusa, ami megadja, hogy az XML állomány milyen jellegű elemét tárolja ez a csomópont. Érdemes megemlítenünk, hogy az XML programkönyvtár szerzői bevett programozói fogás szerint

minden lényegesebb adatszerkezetük második elemében elhelyezik ezt a típusmezőt, így a dinamikus adatterületeken elhelyezett adatok típusa mindig kideríthető.

Az `xmlNode` típusú adatszerkezet `type` mezőjében a következő állandók egyikét találhatjuk:

XML_ELEMENT_NODE A csomópont az XML állomány egy elemét, egy csomópontjának adatait tartalmazza. A memóriában elhelyezett dokumentum legtöbb csomópontja ilyen típusú.

XML_TEXT_NODE A csomópont a dokumentum egy szöveges részét, szövegelemét tartalmazza. Ha például a dokumentumban a

```
<a>
  <b/>
  Helló!
</a>
```

sorokat találjuk, akkor a memóriában elhelyezett adatszerkezetben lesz egy `XML_ELEMENT_NODE` típusú `xmlNode` adatszerkezet az `a` elem számára, és ennek lesz két közvetlen leszármazottja, egy a `b` elem számára, valamint egy `XML_TEXT_NODE` típusú, a szöveg számára. Ez a felépítés nagyon ésszerű, viszont következik belőle, hogy a

```
<a>Helló!</a>
```

részletet ábrázoló adatszerkezetben is két csomópont lesz, egy az `a` elem, egy pedig a szövegtartalom számára, azaz a szöveg nem az elem tulajdonsága, hanem a leszármazottja. Ez természetesen nem csak az XML programkönyvtárban van így, hanem elterjedt vezérelvnek tekinthető.

XML_CDATA_SECTION_NODE A csomópont az XML dokumentum adatblokk típusú elemét ábrázolja. Az adatblokk olyan szövegelem, amelyen belül a tiltott karaktereket nem kell rövidítésükre cserélni, mert azokat az XML értelmezőprogram nem kezeli. Az ilyen szerkezet akkor lehet hasznos, ha az XML állományba hosszabb szöveges értéket szeretnénk tárolni – például egy másik XML állomány tartalmát – és nem szeretnénk a tiltott karakterek cseréjével vésződni.

XML_COMMENT_NODE A csomópont az XML állomány egy megjegyzését tárolja.

name A csomópont neve, ami az `XML_ELEMENT_NODE` típus esetén megegyezik az XML elem nevével.

children Az adott csomópont első leszármazottját jelölő mutató, ha a csomópontnak van leszármazottja, **NULL**, ha nincs.

Az XML programkönyvtár szerzői a fa tárolását bevett szokás szerint kétszeresen láncolt lista segítségével tárolják. Ennél a módszer-nél a szülő egy mutatóval hivatkozik az első és egy mutatóval az utolsó leszármazottjára, a többi leszármazott elhelyezkedéséről viszont nincs információja. Ezeket a leszármazottakat kizárólag azért tudjuk megtalálni, mert kétszeresen láncolt listába vannak rendezve.

A szülő közvetlen leszármazottait – a „testvéreket” (*sibling*, testvér) – tehát listába szervezzük, a lista elején és végén található leszármazottakat pedig nyilvántartásba vesszük a szülő megfelelő mezőjében.

last Az utolsó közvetlen leszármazott a leszármazottak listájában.

parent Hivatkozás a szülőre.

next A következő testvér a listában, ha van ilyen, **NULL**, ha nincs.

prev Az előző testvér, ha létezik, **NULL**, ha nem.

doc A memóriában elhelyezett dokumentumot jelölő mutató, amelyben ez a csomópont található.

ns A névtér mutatója.

content A csomópont szöveges tartalma. Ha a csomópont **type** mezőjének értéke **XML_TEXT_NODE**, akkor magát a szöveget ez a mező jelöli a memóriában.

properties Az elem tulajdonságait tartalmazó listát jelölő mutató. A tulajdonságok tárolására szolgáló adatszerkezetekre a továbbiakban visszatérünk.

line Az állomány sorának száma, aminek az elemzése során ez a csomópont létrejött. A felhasználó számára kiírt üzenetekben érdemes megemlíteni a sor számát is, hogy legalább a minimális esélyt megadjuk számára a hivatkozott rész megtalálására.

23. példa. A következő függvény rekurzívan bejárja a dokumentum csomópontjait és kiírja az egyes csomópontokhoz tartozó szövegértékeket.

```

1 void
2 print_nodes_rec(xmlNodePtr node)
3 {
4     xmlNodePtr      child_node;
```

```

5
6     if (node == NULL)
7         return;
8
9     if (node->type == XML_TEXT_NODE)
10        printf("%s\n", node->content);
11
12    child_node = node->xmlChildrenNode;
13    while (child_node != NULL) {
14        print_nodes_rec(child_node);
15        child_node = child_node->next;
16    }
17 }
```

A függvény a 9. sorban megvizsgálja, hogy az aktuális csomópont szöveget reprezentál-e, ha igen, a 10. sorban kiírja a csomópont szövegértékét. Ezek után a függvény a 12–16. sorok között bejárja a csomópont összes leszármazottját és rekurzív függvényhívással (14. sor) azoknak is kiírja a szöveges értékét.

Az XML programkönyvtár az XML elemek tulajdonságainak tárolására az `xmlAttr` struktúrát használja, a struktúra memóriabeli címét pedig az `xmlAttrPtr` típusú mutatóban tárolja. A programkönyvtár az XML elemek tulajdonságait listába rendezi és az elem adatait tároló adatszerkezet `properties` mezőjével jelöli. Az `xmlAttr` struktúra legfontosabb mezői a következők:

A <code>xmlAttr</code> struktúra	
<code>void *_private</code>	Szabadon használható.
<code>xmlElementType type</code>	Mindig <code>XML_ATTRIBUTE_NODE</code> .
<code>const xmlChar *name</code>	A tulajdonság neve.
<code>struct _xmlNode *children</code>	Az érték csomópont címe.
<code>struct _xmlNode *last</code>	Mindig <code>NULL</code> .
<code>struct _xmlNode *parent</code>	A szülő.
<code>struct _xmlNode *next</code>	A következő tulajdonság.
<code>struct _xmlNode *prev</code>	Az előző tulajdonság.
<code>struct _xmlDoc *doc</code>	A dokumentum.
<code>xmlNs *ns</code>	Névtér.

Amint az a táblázatból is látható, az XML programkönyvtár tulajdonságok tárolására használt `xmlAttr` struktúrája nagyon hasonlít a csomópontok tárolására használt `xmlNode` struktúrájával. Az egyes mezők értelmezése a következő:

164

void *_private E mező a **xmlNode** struktúra azonos mezőjével hasonlóan az alkalmazásprogramozó számára van fenntartva. Arra használhatjuk tehát, amire csak akarjuk.

xmlElementType type Az adatstruktúra típusa, amelynek segítségével azonosíthatjuk a **xmlAttr** típusú adatszerkezetet a memóriában. Itt mindig az **XML_ATTRIBUTE_NODE** értéket találjuk.

const xmlChar *name A tulajdonság nevét hordozó karakterlánc helye a memóriában.

struct _xmlNode *children A tulajdonság értéke. A tulajdonságok értékét az XML programkönyvtár a tulajdonságok értékét **xmlNode** típusú szöveges értékű (a **type** mezőben **XML_TEXT_NODE** értéket hordozó) adatszerkezetekben helyezi el, amelyek címét e mező hordozza.

struct _xmlNode *last Ennek a mezőnek az értéke lényegtelen, csak a hasonlóság érdekében szerepel a struktúrában.

struct _xmlNode *parent A szülő csomópont címe. A szülő az a csomópont, amelyhez ez a tulajdonság tartozik.

struct _xmlAttr *next A szülőcsomópontához tartozó következő tulajdonság címe a memóriában, ha van ilyen, **NULL**, ha nincs.

struct _xmlAttr *prev Az előző tulajdonság címe vagy **NULL**.

struct _xmlDoc *doc A dokumentum címe, amelyhez ez a tulajdonság szerepel.

xmlNs *ns A névtér címe a memóriában.

A következő néhány példa bemutatja a memóriában elhelyezett dokumentum kezelésének néhány jellegzetes lépését.

24. példa. A következő függvény a 23. példa továbbfejlesztett változata, ami a csomópontok bejárása során nem csak azok szöveges értékeit, hanem tulajdonságait is kiírja.

```

1 void
2 print_nodes_rec(xmlNodePtr node)
3 {
4     xmlNodePtr      child_node;
5     xmlAttrPtr      attr_ptr;
6
7     if (node == NULL)
8         return;

```

```

9
10     printf("%s = ", node->name);
11
12     if (node->type == XML_TEXT_NODE)
13         printf("%s\n", node->content);
14     else
15         printf("\n");
16
17     attr_ptr = node->properties;
18     while (attr_ptr != NULL) {
19         printf("  %s = %s\n",
20             attr_ptr->name,
21             attr_ptr->children->content);
22         attr_ptr = attr_ptr->next;
23     }
24
25     child_node = node->xmlChildrenNode;
26     while (child_node != NULL) {
27         print_nodes_rec(child_node);
28         child_node = child_node->next;
29     }
30 }
```

A függvény a 10. sorban kiírja az aktuális csomópont nevét. Az XML programkönyvtár gondoskodik róla, hogy a név címét hordozó mutató soha ne legyen `NULL` értékű, ezért itt nem kell ellenőrzést végeznünk.

A következő lépésben a 12–15. sorokban a név után – ugyanabba a sorba – kiírjuk a csomópontához tartozó szövegértéket, ha úgy találjuk, hogy szöveges értéket hordozó csomópontról van szó. Ha a csomópont nem szöveges, egyszerűen egy újsor karakter írunk ki, hogy a kimenet olvashatóbb legyen.

A függvény a 17–23. sorokban kiírja a csomópontához tartozó tulajdonságok nevét és értékét. Ehhez a 18. sorban kezdődő ciklus segítségével végigjárjuk a tulajdonságokat tároló listát a 17. sorban olvasható kindulóponttól a 22. sor lépésközéig. Ha a csomópontnak nincsen egyetlen tulajdonsága sem, akkor a ciklusmag természetesen egyszer sem fut le.

A tulajdonságok nevét a 20. sor alapján a tulajdonságot tároló adatszerkezetnél, értékét a 21. sor alapján a tulajdonság leszármazottjaként szereplő szöveges csomópont megfelelő mezőjéből olvassuk ki.

A csomópontok rekurzív bejárása – a már ismertetett módon – a függvény 25–29. soraiban látható.

A függvény kimenetének bemutatásához tegyük fel, hogy egy XML állomány tartalmazza a következő sorokat:

```

1  <document type="article" year="2006">
2    <author>
3      <first-name>Lisa</first-name>
4      <last-name>Simpson</last-name>
5    </author>
6    <title>What a Family?!</title>
7  </document>

```

Ha a függvénynek a **document** nevű elemet reprezentáló csomópontot adjuk át paraméterként, a következő sorokat írja a kimenetre:

```

document =
  type = article
  year = 2006
author =
first-name =
text = Lisa
last-name =
text = Simpson
title =
text = What a Family?!

```

Megfigyelhetjük, hogy a függvény a tulajdonságokat két karakterrel beljebb írja, hogy megkülönböztethessük azokat az elemektől, amelyek a sor elején kezdődnek.

Az is jól látható, hogy az elemek csomópontjainak szöveges értékét külön csomópontban tárolja a programkönyvtár. A szöveges értéket hordozó csomópontok neve minden esetben **text**.

25. példa. A következő függvény visszaadja a csomóponthoz tartozó XML elem nevét:

```

1  xmlChar *
2  xml_node_get_name(xmlNodePtr node)
3  {
4      if (node->type != XML_TEXT_NODE)
5          return g_strdup(node->name);
6      else
7          return g_strdup(node->parent->name);
8  }

```

A függvény megvizsgálja, hogy szöveges jellegű csomópontról van-e szó, ha igen, akkor nem a csomópont, hanem a szülő nevét adja vissza.

A függvény a karakterlánc másolására a **g_strdup()** függvényt használja és nem az XML programkönyvtár hasonló célra használható

`xmlCharStrdup()` vagy `xmlStrdup()` függvényét. A választás tulajdonképpen ízlés kérdése.

26. példa. A következő függvény bemutatja hogyan készíthetünk örökölhető tulajdonságokat az XML elemek számára.

```

1  xmlChar *
2  xml_node_get_class(xmlNodePtr node)
3  {
4      xmlAttrPtr      attr_ptr;
5
6      if (node == NULL)
7          return g_strdup("");
8
9      attr_ptr = node->properties;
10     while (attr_ptr != NULL) {
11         if (strcmp(attr_ptr->name, "class") == 0)
12             return g_strdup(attr_ptr->children->content);
13         attr_ptr = attr_ptr->next;
14     }
15
16     return xml_node_get_class(node->parent);
17 }
```

A függvény a 9–14. sorokban megvizsgálja, hogy a csomópontnak van-e **class** nevű tulajdonsága. Ha van, akkor visszaadja annak értékét a 12. sorban, miután lemásolta.

Ha a függvény nem találta meg a keresett tulajdonságot, újra próbálkozik a csomópont szülőjével, azaz a **class** tulajdonság örökölhető. Ha viszont a keresés során a függvény eléri a dokumentum gyökerét, a 7. sorban visszaadja az alapértelmezett értéket, ami az üres karakterlánc másolata. (Nyilvánvaló, hogy azért másoljuk le az üres karakterláncot, hogy a hívónak ne kelljen azon gondolkodnia, hogy felszabadítsa-e a memóriaterületet.)

6.2.7. A dokumentum módosítása

Ahogy már láttuk az `xmlNewDocNode()`, `xmlNewChild()` függvényekkel a dokumentumban csomópontokat hozhatunk létre, amelyeket az `xmlSetProp()` függvénnyel új tulajdonságokkal láthatunk el, így a dokumentumot lépésről-lépésre felépíthetjük. A következő oldalakon olyan függvényeket veszünk sorra, amelyekkel a módosítás bonyolultabb lépéseit is elvégezhetjük.

Előfordulhat például, hogy a dokumentum faszerkezetét alulról felfelé szeretnénk felépíteni, mert a program algoritmusának ez a természetes sorrendje. (A nyelvtani elemzők legelterjedtebb csoportja például pontosan így működik, azaz egy fát épít fel alulról felfelé.) Ilyen esetben is használhatjuk a következő függvényeket:

`xmlNodePtr xmlNewNode(xmlNsPtr névtér, const xmlChar *név);` A függvénnyel új csomópontot hozhatunk létre anélkül, hogy meghatároznánk, hogy a dokumentum melyik részére fog kerülni.

A függvény első paramétere a használni kívánt névteret jelöli a memóriában vagy `NULL`, ha nem kívánunk névteret meghatározni.

A függvény második paramétere a létrehozandó csomópont nevét jelöli a memóriában, értéke nem lehet `NULL`. A függvény a névről dinamikus területre elhelyezett másolatot készít, így a karakterláncot nem kell megőriznünk. (Az `xmlNewNodeEatName()` hasonlóan működik, de felhasználja a nevet.)

A függvény visszatérési értéke a létrehozott csomópontot jelöli a memóriában. Az új csomópont `type` mezőjének értéke `XML_ELEMENT_NODE`, azaz az új csomópont az XML állomány egy elemét reprezentálja.

`xmlNodePtr xmlNewText(const xmlChar *szöveg);` A függvény új szöveges csomópontot hozhatunk létre.

A függvény paramétere a csomópontban tárolt szöveget jelöli a memóriában. A függvény a szövegről másolatot készít dinamikus memóriaterületre, így azt nem kell megőriznünk.

A függvény visszatérési értéke a létrehozott új csomópontot jelöli a memóriában.

E függvény kapcsán érdemes megemlítenünk, hogy nem szerencsés, ha „kézzel”, az XML programkönyvtár megkerülésével próbálunk meg szöveges csomópontot létrehozni, mert a programkönyvtár a szöveges csomópontok nevének (`text`) tárolására egyetlen memóriaterületet használ és ki is használja, hogy az összes szöveges csomópontok neve mindig ugyanarra a memóriaterületre mutat.

`xmlNodePtr xmlNewComment(const xmlChar *szöveg);` A függvény segítségével új, megjegyzést tartalmazó csomópontot hozhatunk létre, amelyet később a dokumentum megfelelő részére elhelyezhetünk.

A függvény paramétere a megjegyzés szövegét jelöli a memóriában. A függvény erről a szövegről másolatot készít, így azt nem kell megőriznünk.

A függvény visszatérési értéke az új csomópontot jelöli a memóriában.

`xmlNodePtr xmlNewCDataBlock(xmlDocPtr doc, const xmlChar *szöveg, int hossz);` A függvény segítségével új, adatblokk típusú csomópontot hozhatunk létre. Az adatblokk olyan csomópont, amelyen belül a tiltott karaktereket nem kell rövidítéseikre cserélni.

A függvény első paramétere a dokumentumot jelöli a memóriában. Ha itt `NULL` értéket adunk meg, a csomópontban a dokumentum nyilvántartása helyreállítódik, amikor a csomópontot a megfelelő szülő megadásával elhelyezzük a dokumentumban.

A függvény második paramétere az új csomópont szövegét jelöli a memóriában, harmadik paramétere pedig megadja, hogy legfeljebb milyen hosszon készítsen a függvény másolatot a szövegről.

A függvény visszatérési értéke a létrehozott új csomópontot jelöli a memóriában.

`xmlNodePtr xmlAddChild(xmlNodePtr szülő, xmlNodePtr új);` A függvénnyel új elemet helyezhetünk el a dokumentumban, az elem helyeként a leendő szülőcsomópont meghatározásával. A függvény hívásakor fel kell készülnünk arra, hogy az az új elem bizonyos esetekben megsemmisíti.

A függvény első paramétere a leendő szülőt, második paramétere pedig az elhelyezendő új elemet jelöli a memóriában.

Ha az új csomópont szöveges és a szülőcsomópont utolsó gyermeke szintén az, akkor a függvény az új csomópont szövegét az utolsó gyermekcsomópont szövegéhez hozzámásolja és az új csomópontot felszabadítja. Ilyen esetben a függvény a szülő utolsó gyermekének címét adja vissza.

Ha a szülő szöveges típusú, a függvény nem gyermekként helyezi el az új csomópontot – a szöveges csomópontoknak nem lehet gyermekük –, hanem megkísérli a szülő szövegértékét a gyermek szövegértékével bővíteni és szintén felszabadítja az új elemet. Ilyen esetben a függvény a szülő címét adja vissza.

Ha a gyermekcsomópont csomóponttulajdonságot ír le – a `type` mezőjének értéke `XML_ATTRIBUTE_NODE` –, a függvény nem a gyermekek közé, hanem a tulajdonságok közé illeszti be, de mielőtt megtenné törli az esetleges azonos nevű, létező tulajdonságot. A visszatérési érték ilyen esetben az új csomópont címe.

Minden más esetben a függvény beilleszti az utolsó gyermek után az új csomópontot és visszaadja annak címét.

`xmlNodePtr xmlAddNextSibling(xmlNodePtr testvér,
xmlNodePtr új);` A függvény segítségével új elemet helyezhetünk el a dokumentumban a testvérének megadásával.

A függvény első paramétere megadja melyik csomópont után szeretnénk beilleszteni a csomópontot, a második paramétere pedig az új csomópontot jelöli a memóriában.

A függvény visszatérési értéke az új elem memóriacíme, de ennél a függvéynél is számítanunk kell arra, hogy a szöveges csomópontokat a programkönyvtár egymásbaolvastja, az új elemet megsemmisíti és annak az elemnek a címét adja vissza, amelyek összeolvasztotta az új elemet.

`xmlNodePtr xmlAddPrevSibling(xmlNodePtr testvér,
xmlNodePtr új);` A függvény használata és működése megegyezik az `xmlAddNextSibling()` függvény használatával és működésével, azzal a különbséggel, hogy ez a függvény a megadott csomópont elé illeszti be a csomópontot.

`xmlNodePtr xmlAddSibling(xmlNodePtr testvér, xmlNodePtr új);` E függvény használata és működése megegyezik a `xmlAddNextSibling()` függvény használatával és működésével, azzal a különbséggel, hogy ez a függvény a testvércsomópontok listájának végére, utolsó testvérként illeszti be az új csomópontot.

6.3. Az XPath eszköztár

Az XPath egy egyszerű nyelv, amelyet kimondottan az XML adatszerkezetekben való keresésre fejlesztettek ki. Az XPath nyelv segítségével kijelölhetjük, megszámálhatjuk, átalakíthatjuk az XML nyelven leírt adatok elemeit – a csomópontokat, csomóponttulajdonságokat, a csomópontokhoz tartozó szöveges értékeket. Az XPath tehát tulajdonképpen egy fejlett lekérdezőnyelv, ami igen hatékonyan használható az XML állományok kezelésére.

Az XML programkönyvtár támogatja az XPath nyelvet, így magas szintű eszköztárat biztosít a programozó számára a memóriában tárolt XML dokumentumokban való keresésre. Mielőtt azonban az XML programkönyvtár ide vonatkozó függvényeit megismernénk, érdemes néhány szót ejteni magáról az XPath nyelvről.

Az XPath az XML csomópontokon a UNIX állományrendszerhez hasonlóan értelmezi az ösvény fogalmát, amennyiben a gyökércsomópont neve a `/` jel és a csomópontok nevéből ösvényt szintén a `/` jellel készíthetünk. Valójában – ahogyan az XPath elnevezés is jelzi (*path*, ösvény) – a nyelv legegyszerűbb kifejezései egy ösvényt adnak meg az XML faszerkezetben.

A `/a/b` például a gyökérelemből elérhető a `a` elemen belül található `b` elemet jelöli. A UNIX könyvtárszerkezeténél megszokott jelöléshez hasonló a `..` jelentése is, ami az XPath nyelvben a szülőelemet jelöli.

A csomópontokból készült ösvény esetében azonban gondolnunk kell arra, hogy a UNIX állományrendszerrel szemben az XML állomány adott helyén több azonos nevű csomópont lehet. Az XPath az egyszerű ösvény esetében mindig az első illeszkedő nevű csomópontot választja. A

```
/database/document/title
```

kifejezés segítségével például egy csomóponthalmazt azonosít, amelyben a gyökérelemben található összes `document` elem összes `title` eleme beletartozik.

Az ösvény megadásakor hivatkozhatunk a csomópontok értékére is. Ehhez a csomópont neve elé egyszerűen csak a `@` karaktert kell írunk, jelezve, hogy nem csomópontról, hanem csomóponttulajdonságról van szó. A `/document/@version` például a `document` nevű csomópont `version` nevű tulajdonságát jelöli.

Vegyük észre, hogy a `/@version` kifejezésnek például nem sok értelme van, hiszen a `/` csomópont nem létező XML elem – csak a faszerkezet kezdőpontjának jelölése – és így tulajdonságai sem lehetnek. Ugyanígy a `/document/@version/title` kifejezés is értelmetlen, hiszen a tulajdonságokon belül nem lehet újabb csomópontokat elhelyezni.

A `*` helyettesítőkaraktert mind a csomópontok, mind pedig a tulajdonságok neveinek helyettesítésére használhatjuk. A

```
/database/*/@*
```

kifejezés például olyan csomóponthalmazt jelöl, amelybe beletartozik a `database` nevű gyökerelem összes leszármazottjának összes csomóponttulajdonsága. Nem használhatjuk viszont a `*` karaktert a csomópont vagy tulajdonság nevének karaktereivel keverve, azaz a `*a` és a `@a*` egyaránt hibás.

Az XPath nyelv kifejezőerejét nagymértékben megnöveli a `[]` zárójelpár, aminek a segítségével a kiválasztott csomópontok listáját szűrhetjük. A szögletes zárójelek közt hivatkozhatunk a csomópontok és a csomóponttulajdonságok értékére, használhatunk műveleti jeleket és függvényeket, így az XPath kifejezés segítségével igen pontosan megfogalmazhatjuk, hogy pontosan milyen csomópontokat keresünk. A

```
/database/document[@type='article']/author/first-name  
/database/document/author[first-name = 'Bart']/../title
```

kifejezésekkel például lekérdezhetjük azoknak a szerzőknek a keresztnévét, akik cikk típusú dokumentumot írtak, illetve azoknak a dokumentumoknak a címét, amelyeknek szerzője a Bart keresztnévre hallgat. A függvények segítségével a kifejezések tovább finomíthatók. A


```
/database/document[contains(author, 'Hom')]/title
/database/document[position()=2]/author
```

kifejezések segítségével például kiválaszthatjuk azoknak a dokumentumoknak a címét, amelyek szerzőjének a nevében szerepel a **Hom** karakterlánc, illetve a sorrendben második dokumentum szerzőjét. Ez utóbbi kifejezést, az azonos nevű csomópontok sorában elfoglalt hely megadását rövidíthetjük a

```
/database/document[2]/author
```

formában.

A szűrőkifejezések kapcsán fontos, hogy szót ejtsünk a csomópontok értékéről. Az XPath kifejezésekben a csomópontok szöveges értéke – az érték, amelyet a csomópont nevéhez rendelünk – az összes leszármazott csomópont szöveges értékének és magának a csomópontnak a szöveges értékének egymás után való másolásával kapható meg. Ha például egy XML állomány részlete

```
<a><b>egy</b><c>kettő</c>három</a>
```

akkor a **a** csomópontnévhez az **egyketthárom** szöveges érték tartozik. Ha kizárólag az adott csomóponthoz tartozó szöveges értéket akarjuk lekérdezni, akkor a **text()** XPath függvényt kell használnunk. A

```
/database/document[position()=1]/text()
```

kifejezés például az első **document** nevű csomópontoz tartozó szöveget adja, amibe nem értendő bele a leszármazott csomópontok értéke.

Ahogy látható a függvényeket nem csal a **[]** jelek közt használhatjuk. A különféle XPath függvényeknek különféle visszatérési értékei vannak, így előfordulhat, hogy az XPath kifejezés nem csomópontot, nem is csomópontok halmazát adja, hanem például számértéket. A

```
count(/database/document)
```

számértékként megadja, hogy az adatbázisban hány dokumentumról tárolunk adatokat.

Az XPath függvényekre a későbbiekben még visszatérünk, előbb azonban bemutatjuk a legfontosabb függvényeket, amelyek az XPath kifejezések kiértékeléséhez szükségesek.

6.3.1. Egyszerű XPath kifejezések kiértékelése

Az XML programkönyvtár következő függvényei segítségével az XPath kifejezések kiértékelhetők, a memóriában elhelyezett XML dokumentumban tárolt adatok egyszerűen lekérdezhetők:

`xmlChar *xmlGetNodePath(xmlNodePtr node);` A függvény segítségével az adott csomópontoz tartozó ösvényt állapíthatjuk meg.

A függvény argumentuma a csomópontot jelöli a memóriában, visszatérési értéke pedig egy olyan dinamikusan foglalt memóriaterületre mutat, amelyben olyan XPath kifejezés van elhelyezve, aminek eredménye éppen az argumentum által kijelölt csomópont. A függvény hiba esetén `NULL` értéket ad vissza.

`xmlXPathContextPtr xmlXPathNewContext(xmlDocPtr doc);` Az XML programkönyvtár az XPath kifejezéseket az XPath környezet (*context*, szövegekörnyezet, kontextus) figyelembevételével hajtja végre. Mielőtt tehát XPath kifejezések kiértékeléséhez fognánk, legalább egy XPath környezetet létre kell hoznunk ezzel a függvénnyel.

A függvény argumentuma a memóriában elhelyezett dokumentumot, visszatérési értéke pedig a létrehozott XPath környezetet jelöli a memóriában.

`void xmlXPathFreeContext(xmlXPathContextPtr kontextus);` A függvény segítségével az XPath környezetet semmisíthetjük meg, ha már nincs szükségünk rá.

A függvény argumentuma a megsemmisítendő környezetet jelöli a memóriában.

`xmlXPathObjectPtr xmlXPathEvalExpression(const xmlChar *kifejezés, xmlXPathContextPtr kontextus);` A függvény segítségével a karakterláncként megadott XPath kifejezést az adott környezet alapján kiértékelhetjük, azaz kiszámíthatjuk az értékét.

A függvény első paramétere a karakterláncként ábrázolt XPath kifejezést, második paramétere pedig az XPath környezetet – amelyet az `xmlXPathNewContext()` függvénnyel hoztunk létre – jelöli a memóriában.

A függvény visszatérési értéke az XPath kifejezés kiértékelésének eredményét tároló összetett adatszerkezetet jelöli a memóriában, ha a paraméterként meghatározott XPath kifejezés hibátlan, `NULL` érték, ha a kifejezésben nyelvtani hiba volt. Mivel a kiértékelés eredménye szöveges, szám jellegű, sőt csomóponthalmaz is lehet, az eredmény kezelése nem mindig egyszerű.

Az XPath kifejezések kiértékelése során kapott `xmlXPathObjectPtr` típusú adatszerkezet részletes értelmezésére a későbbiekben visszatérünk.

`void xmlXPathFreeObject(xmlXPathObjectPtr obj);` A függvény segítségével az XPath kifejezés kiértékelésekor kapott

`xmlXPathObjectPtr` típusú mutatóval jelzett adatszerkezetet felszabadíthatjuk.

A függvény paramétere a felszabadítandó adatszerkezetet jelöli a memóriában.

`xmlChar *xmlXPathCastToString(xmlXPathObjectPtr val);` A függvény segítségével az XPath kifejezés eredményét karakterlánc-ként kérdezhetjük le.

A függvény paramétere az XPath kifejezés kiértékelésekor kapott mutató – a `xmlXPathEvalExpression()` visszatérési értéke – visszatérési értéke pedig az XPath lekérdezés szöveges eredménye vagy `NULL` érték hiba esetén.

Ez a függvény nem `NULL` értéket ad, ha a kérdéses XPath kifejezésnek nem szöveges az értéke – azaz ha például a keresett csomópont nem létezik –, hanem egy 0 hosszúságú karakterláncot hoz létre és annak a mutatóját adja visszatérési érték-ként.

Ha az XPath kifejezés értéke csomóponthalmaz, akkor a függvény a csomópontok közül csak a legelsőket veszi figyelembe, annak a szöveges értékét adja.

A dokumentáció szerint a függvény visszatérési értéke dinamikusan foglalt memóriaterületre mutat – amelyet fel kell szabadítanunk az `xFree()` függvénnyel – ha az XPath kifejezés eredménye nem szöveges jellegű volt. Az XML programkönyvtár forrását átböngészve viszont arra kell ráébrednünk, hogy az `xmlXPathCastToString()` mindig dinamikusan foglalt memóriaterületet ad vissza, amit mindig fel kell szabadítanunk. Ez megkönnyíti a munkánkat, és egyben jelzi, hogy az XML programkönyvtár dokumentációja sajnos nem csak szűkszavú, de helyenként hibás is.

Ez a lekérdezés értékét szöveges alakra hozza akkor is, ha az eredetileg szám jellegű volt, használhatjuk tehát, ha a számra szövegként van szükségünk.

`double xmlXPathCastToNumber(xmlXPathObjectPtr val);` A függvény segítségével az XPath kifejezés kiértékelési értékét kérdezhetjük le szám formában.

A függvény paramétere a kifejezés kiértékelésének eredményét jelöli a memóriában, visszatérési értéke pedig a kifejezés értéke szám formában.

`int xmlXPathCastToBoolean(xmlXPathObjectPtr val);` A függvény segítségével az XPath kifejezés értékét kérdezhetjük le logikai érték-ként.

A függvény paramétere a kifejezés eredményét jelöli a memóriában, visszatérési értéke pedig 1 vagy 0, aszerint, hogy az érték logikai igaz, vagy hamis értéket képvisel.

A bemutatott függvények segítségével, a `xmlXPathObjectPtr` mutató által kijelölt adatszerkezet szerkezetének pontos ismerete nélkül is elvégezhetünk lekérdezéseket. Ezt mutatja be a következő példa.

27. példa. A következő függvény az XPath kifejezés kiértékelésének lehető legegyszerűbb módját mutatja be, csak a legszükségesebb függvények hívásával.

```

1 void
2 xpath_search(xmlDocPtr document,
3               const gchar *path)
4 {
5     xmlXPathContextPtr xpath_ctx;
6     xmlXPathObjectPtr xpath_obj;
7     xmlChar             *string;
8
9     xpath_ctx = xmlXPathNewContext(document);
10
11     xpath_obj = xmlXPathEvalExpression(path, xpath_ctx);
12     string = xmlXPathCastToString(xpath_obj);
13     g_message("%s='%s'", path, string);
14
15     xmlFree(string);
16     xmlXPathFreeObject(xpath_obj);
17     xmlXPathFreeContext(xpath_ctx);
18 }
```

Az XPath kifejezés kiértékelése és a szöveges érték kinyerése könnyen végigkövethető a függvény olvasásával. A függvény hívása során a 22. példa XML adatszerkezete esetében a függvény a következő sorokat írta ki:

```

1  ** Message: /root/if/condition='8 < 9'
2  ** Message: /root/if/@type='numeric'
```

Figyeljük meg, hogy a lekérdezés során az eredeti karakterláncokat kaptuk vissza, a tiltott karakterek rövidítéseit az XML programkönyvtár az eredeti karakterekre cserélte le.

6.3.2. Az összetett eredményű XPath kifejezések

Az XPath kifejezések eredményét az XML programkönyvtár az `xmlXPathObject` típusú struktúrában helyezi el, a struktúra címét pedig az `xmlXPathObjectPtr` típusú mutatóban adja át. A struktúra legfontosabb mezői a következők:

Az <code>xmlXPathObject</code> struktúra	
<code>xmlXPathObjectType type</code>	Az eredmény típusa.
<code>xmlNodeSetPtr nodesetval</code>	Az elemhalmaz érték.
<code>int boolval</code>	A logikai érték.
<code>double floatval</code>	A szám jellegű érték.
<code>xmlChar *stringval</code>	A karakterlánc érték.

A struktúra `type` eleme meghatározza, hogy az eredmény milyen jellegű. Itt – többek között – a következő állandók egyikét találhatjuk meg:

`XPATH_UNDEFINED` Az eredmény ismeretlen típusú, nem hordoz értéket.

`XPATH_NODESET` Az eredmény típusa csomóponthalmaz, értékét a `nodesetval` mező jelöli a memóriában.

`XPATH_BOOLEAN` Az eredmény logikai típusú, értéke a `boolval` mezőben található.

`XPATH_NUMBER` Az eredmény szám típusú, értéke a `floatval` mezőben található.

`XPATH_STRING` Az eredmény karakterlánc típusú, értéke `stringval` mezőben található mutatóval jelzett memóriaterületen található.

`XPATH_XSLT_TREE` Az eredmény típusa csomóponthalmaz, értéke a `nodesetval` mezőben található.

Ha az XPath kifejezés eredménye csomóponthalmaz, akkor az eredményül kapott `xmlXPathObject` adatszerkezet `xmlNodeSetPtr` típusú mutatója egy `xmlNodeSet` adatszerkezetet jelöl a memóriában. Az `xmlNodeSet` – ahogyan a neve is mutatja – csomóponthalmazok tárolására használható adatszerkezet, ami a következő mezőkből áll:

Az <code>xmlNodeSet</code> struktúra	
<code>int nodeNr</code>	A csomópontok száma.
<code>int nodeMax</code>	A foglalt terület mértéke.
<code>xmlNodePtr *nodeTab</code>	A csomópontok címét tároló tömb címe.

Ezen ismeretek birtokában könnyen írhatunk olyan függvényt, ami az XPath kifejezés eredményeként kapott csomóponthalmazt végigjárja. Ezt mutatja be a következő példa.

28. példa. A következő függvény argumentumként az XPath kifejezés eredményét kapja, az abban tárolt csomópontok mindegyikét végigjárja, az értéküket kiíró függvényt meghívja.

```

1  void
2  print_nod eset(xmlXPathObjectPtr xpath_obj)
3  {
4      xmlNodeSetPtr cur;
5      int n;
6
7      g_assert(xpath_obj->type == XPATH_NODESET);
8
9      cur = xpath_obj->nod esetval;
10     for (n = 0; n < cur->nodeNr; n++)
11         g_message("Val: '%s'",
12                 xmlNodeGetContent(cur->nodeTab[n]));
13 }

```

A függvényt csak olyan `xmlXPathObjectPtr` mutatóval hívhatjuk, ami csomóponthalmaz jellegű adatokat hordoz és ezt a függvény a 7. sorban ellenőrzi is.

A függvény a 10–12. sorokban található ciklussal végigjárja az argumentum által jelölt adatszerkezetben – az XPath kifejezés eredményében – található összes csomópontot és kiírja ezek szöveges értékét.

6.3.3. Az XPath műveleti jelek és függvények

Az XPath kifejezésekben az XML programkönyvtár szöveges és szám jellegű állandók, műveleti jelek – operátorok – és függvények használatát is lehetővé teszi. A szöveges állandókat a `" "` vagy a `' '` jelek közé kell írunk, a szám jellegű állandókat pedig a szokásos formában adhatjuk meg. Mivel a C programozási nyelv karakterláncait a `" "` jelekkel kell tárolnunk, szerencsés, ha az XPath karakterlánc típusú állandóihoz a `' '` jeleket használjuk (például `concat('hello', ' ', 'vilag')`). Az XPath kifejezés eredményének típusa természetesen magától a kifejezéstől függően alakul.

A szám jellegű értékeken a szokásos módon használhatjuk a négy alapművelet jelét – `+`, `-`, `*`, `/` – és a `( )` zárójeleket.

"boolean", "ceiling", "count", "concat", "contains", "id", "false", "floor", "last", "lang", "local-name", "not", "name", "namespace-uri", "normalize-space", "number", "position", "round", "string", "string-length", "starts-with", "substring", "substring-before", "substring-after", "sum", "true", "translate",

7. fejezet

Többablakos alkalmazások

E fejezetben azokról az eszközökről lesz szó, amelyek segítségével több ablakot használó alkalmazásokat készíthetünk. A bemutatott eszközök szükségesek az ablakok nyitásához és zárásához, illetve ahhoz, hogy az egyes ablakokhoz tartozó programrészek adatokat cseréljenek egymással.

7.1. Az ablakok megnyitása, bezárása

Ha a Glade segítségével több ablakot készítünk, a program fordítása után az összes ablak megjelenik a képernyőn. A legtöbb esetben természetesen nem kívánjuk megnyitni az összes ablakot az alkalmazás indításakor, így nyilvánvaló, hogy valamit tennünk kell.

A Glade alapértelmezés szerint a projekt alapkönyvtárában található [src/](#) alkönyvtárban lévő [main.c](#) állományban helyezi el az ablakok megnyitásáért felelős programrészt. A következő sorok ezt a részt ábrázolják.

```
1  /*
2   * The following code was added by Glade to create one
3   * of each component (except popup menus), just so
4   * that you see something after building the project.
5   * Delete any components that you don't want shown
6   * initially.
7   */
8  window1 = create_window1 ();
9  gtk_widget_show (window1);
10 window2 = create_window2 ();
11 gtk_widget_show (window2);
```


180

A programrészlet magyarázatának fordítása:

```

1  /*
2   * A következő programrészt a Glade azért hozta létre,
3   * hogy minden képernyőelemből egy megjelenjen (kivéve
4   * a lebegő menüket), és legyen valami a képernyőn a
5   * program létrehozása után. Törölje ki bármelyik
6   * elemet amelyet kezdetben nem akar megjeleníteni.
7   */

```

A programrészletből megtudhatjuk, hogy a Glade ezt az állományt nem írja felül, ezért nyugodtan módosíthatjuk.

Azt is megtudhatjuk, hogy hogyan tudunk új ablakot nyitni. A Glade minden ablak számára létrehoz egy függvényt, ami létrehozza az ablakot. Az ablakot létrehozó függvény neve szemmel láthatóan tartalmazza magának az ablaknak a nevét. A példában a `window1` és `window2` nevű ablakok létrehozására a `create_window1()`, illetve `create_window2()` függvények használhatók. A függvények visszatérési értéke `GtkWidget *` típusú.

Az ablakok létrehozására, megjelenítésére és bezárására használhatók a következő függvények.

```
GtkWidget *create_xxx(void);
```

Az ilyen névvel ellátott függvényeket a Glade hozza létre az `interface.c` állományban. A függvények deklarációja az `interface.h` állományban vannak.

A függvények létrehozzák az adott nevű ablakot (az ablak neve az „xxx” helyén található) és az ablakban található összes képernyőelemet. Az ablak megjelenítéséről azonban a függvények nem gondoskodnak, azaz az elkészített ablak nem jelenik meg a képernyőn, amíg erről külön nem gondoskodunk.

Az ablakokat létrehozó függvényeket többször is meghívhatjuk egymás után, hogy több egyforma ablakot hozzunk létre.

```
void gtk_widget_show(GtkWidget *elem);
```

Amint azt már a 3.3.2. oldalon láthattuk, e függvény segítségével megjeleníthetjük a képernyőelemeket a képernyőn.

Az ablakok kapcsán annyit kell még tudnunk erről a függvényről, hogy a Glade által létrehozott, ablakok létrehozására szolgáló függvények a Glade *tulajdonságok* ablakának *látható* kapcsolója alapján meghívják ezt a függvényt az ablak belső képernyőelemeinek megjelenítésére. Amikor tehát a `gtk_widget_show()` függvénnyel megjelenítjük magát az ablakot, megjelennek a hozzá tartozó képernyőelemek is.

A függvény paramétere azt a – már létrehozott – képernyőelemet jelöli a memóriában, amelyet meg kívánunk jeleníteni.

`void gtk_widget_show_all(GtkWidget *elem);` A függvény működése megegyezik a `gtk_widget_show()` függvény működésével, de a képernyőelemen található belső képernyőelemeket is megjeleníti, akkor is, ha azok megjelenítéséről még nem gondoskodtunk.

Az Glade segítségével készített ablakok megjelenítésére ezt a függvényt általában nem szerencsés használni, mivel feleslegesen járja végig a belső képernyőelemeket, amelyeket a Glade által készített függvény már megjelenített és amelyek csak azért nem látszanak a képernyőn, mert maga az ablak nem látszik.

`void gtk_widget_destroy(GtkWidget *elem);` Amint azt a 3.3.2. oldalon láttuk, ezt a függvény képernyőelemek megsemmisítésére használhatjuk. Ha valamelyik ablakot be akarjuk zárni, akkor általában nem egyszerűen „eldugjuk” őket a felhasználók elől (lásd a `gtk_widget_hide()` függvényt a 3.3.2. oldalon), hanem megsemmisítjük a teljes tartalmukkal együtt e függvény segítségével.

A függvény paramétere a megsemmisítendő képernyőelemet jelöli a memóriában.

Készítsünk a Glade segítségével egy projektet, amely két ablakot tartalmaz! Az első ablak tartalmazzon egy nyomógombot a második ablak megnyitására, a második ablak pedig tartalmazzon egy nyomógombot a saját ablak bezárására. Az alkalmazást a 7.1. ábra mutatja be.

A következő példa bemutatja hogyan tudjuk a visszahívott függvényekben a megfelelő műveleteket megvalósítani.

29. példa. A következő programrészletben láthatjuk hogyan tudunk új ablakot nyitni és ablakot bezárni, illetve hogyan tudunk a programból kilépni a visszahívott függvények segítségével.



7.1. ábra. Többablakos alkalmazás

```

1  /*
2   * Új ablak nyitása más ablakban található
3   * nyomógomb segítségével.
4   */
5  void
6  on_open_clicked(GtkButton *button,
```

182

```

7           gpointer    user_data)
8   {
9       GtkWidget *window2;
10
11       window2 = create_window2();
12       gtk_widget_show(window2);
13   }
14
15   /*
16    * Kilépés a programból.
17    */
18   void
19   on_exit_clicked(GtkButton *button,
20                  gpointer    user_data)
21   {
22       exit(EXIT_SUCCESS);
23   }
24
25   /*
26    * Ablak bezárása az ablakban található nyomógomb
27    * segítségével.
28    */
29   void
30   on_close_clicked(GtkButton *button,
31                   gpointer    user_data)
32   {
33       GtkWidget *window2;
34       window2 = lookup_widget(GTK_WIDGET(button),
35                              "window2");
36       gtk_widget_destroy(window2);
37   }
38

```

A programrészletben látható visszahívott függvények működése könnyedén megérthető a bemutatott ismeretek alapján.

7.2. A függvényhívással létrehozható ablakok

A következő néhány oldalon olyan ablakokról lesz szó, amelyeket nem a Glade segítségével, hanem a megfelelő függvények hívásával hozunk létre és kezelünk. Ezek a segédeszközök egyszerűbbé teszik a munkánkat, hiszen nem kell nekünk megtervezni az ablakokat, azokat a GTK+

programkönyvtár tartalmazza.

7.2.1. Az üzenet-ablakok

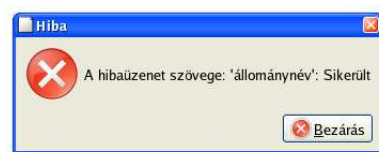
Sokszor előfordul, hogy nem akarunk bonyolult ablakot készíteni, csak egy egyszerű üzenetet szeretnénk megjeleníteni a felhasználó számára. Természetesen megtehetjük, hogy a `printf()` vagy az `fprintf()` függvény segítségével a szabványos kimenetre vagy a szabványos hibacsatornára küldünk egy üzenetet, de ezt a felhasználó nem látja, ha a programunkat nem a parancssorból indította el, hanem egy menü vagy ikon segítségével.

Az is megoldást jelenthet, ha az üzenet számára készítünk egy egyszerű ablakot. Ha a Glade segítségével készítünk egy ablakot, amelyben egy címke és néhány nyomógomb van, az üzenetet el tudjuk helyezni a címkében. Ez azonban kissé bonyolult és fárasztó módszer, főleg, ha az üzeneteinket többféle formában szeretnénk megjeleníteni.

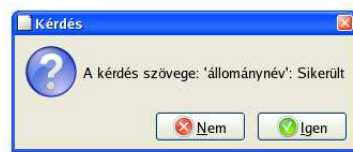
Kényelmes és tetszetős megoldást jelenthet a GTK+ üzenetek megjelenítésére szolgáló ablakainak használata. Ezek az üzenet-ablakok néhány egyszerű függvény segítségével megjeleníthetők és eltávolíthatók és mivel többféle formában állnak a rendelkezésünkre, mindig az adott üzenet jellegéhez igazíthatók. A GTK+ segítségével üzenetek megjelenítésére használható üzenet-ablakokat láthatunk a 7.2. és 7.3. ábrákon. Figyeljük meg, hogy az üzenet jellegének megfelelő ablakcím és ikon jeleníthető meg és az üzenetnek megfelelően egy vagy több nyomógomb is megjeleníthető az ablakokban. Ha több nyomógomb is található az ablakban, természetesen könnyen ki tudjuk deríteni, hogy a felhasználó melyik nyomógombot használta az ablak bezárására.

Tudnunk kell, hogy az üzenet-ablakok modálisak, azaz amíg üzenet ablak látható a képernyőn, addig a felhasználó nem tudja az alkalmazásunk egyéb elemeit használni. Ez roppant hasznos, hiszen így a felhasználó nem tud továbblépni, amíg az adott üzenetet nem zárja le, amíg az adott kérdésre nem válaszolt. Az üzenet-ablakok e tulajdonsága biztosítja a programozó számára, hogy a program adott pontján a válasz már biztosan elérhető legyen, ami egyértelműen megkönnyíti a program megírását.

Az üzenet-ablakok használata közben a következő függvények és állandók a legfontosabbak.



7.2. ábra. A hibaüzenet ablak



7.3. ábra. A kérdés ablak

```
GtkWidget *gtk_message_dialog_new(GtkWindow *szülő,
    GtkDialogFlags kapcsolók, GtkMessageType típus,
    GtkButtonsType gombok, const gchar *formázószöveg,
    ...);
```

A függvény segítségével új üzenet-ablakot készíthetünk a megadott formában, a megadott szöveggel.

A függvény számára a `printf()` függvényhez hasonló módon – formázószöveggel és tetszőleges számú argumentummal – adhatjuk meg az ablakban megjelenítendő szöveget, ezért ezt a függvényt igen könnyen használhatjuk egyszerű üzenetek megjelenítésére grafikus felhasználói felületen.

A függvény argumentumai és visszatérési értéke a következők:

visszatérési érték A függvény által létrehozott ablakot jelöli a memóriában.

szülő A létrehozandó ablak szülőablakát jelöli a memóriában. Az üzenet ablakok mindig rendelkeznek egy szülőablakkal, amely az alkalmazásunk valamelyik ablaka.

kapcsolók Az ablak viselkedését befolyásoló kapcsolók. Itt általában csak a `GTK_DIALOG_DESTROY_WITH_PARENT` állandót adjuk meg, amely arra ad utasítást, hogy a létrehozandó ablakot meg kell semmisíteni, ha a szülőablak megsemmisül.

típus A létrehozandó üzenet-ablak típusa, amely meghatározza az ablak címét és az abban található ikon kinézetét. Itt a következő állandók egyikét használhatjuk:

`GTK_MESSAGE_INFO` Tájékoztató jellegű ablak.

`GTK_MESSAGE_WARNING` Figyelmeztetés jellegű ablak.

`GTK_MESSAGE_QUESTION` Kérdést megfogalmazó ablak.

`GTK_MESSAGE_ERROR` Hibaüzenetet hordozó ablak.

gombok Meghatározza, hogy milyen nyomógombok jelenjenek meg a létrehozott ablakban. Itt a következő állandók egyikét használhatjuk:

`GTK_BUTTONS_OK` Egy darab „OK” feliratú nyomógomb.

`GTK_BUTTONS_CLOSE` Egy darab „bezár” feliratú nyomógomb.

`GTK_BUTTONS_CANCEL` Egy darab „mégsem” feliratú nyomógomb.

`GTK_BUTTONS_YES_NO` Egy „igen” és egy „nem” feliratú nyomógomb.

`GTK_BUTTONS_OK_CANCEL` Egy „OK” és egy „mégsem” feliratú nyomógomb.

A nyomógombok a felhasználó által beállított nyelven megformázott szöveget és a felhasználó által beállított témának megfelelő ikont tartalmazzák.

formázószöveg Ez a paraméter formájában és jelentésében megegyezik a `printf()` szabványos könyvtári függvénynél használt formázószöveggel, meghatározva az üzenet ablakban megjelenő szöveget.

... A formázószövegben hivatkozott értékek sora, ahogyan azt a `printf()` függvénynél is megfigyelhettük.

```
gint gtk_dialog_run(GtkDialog *ablak);
```

A függvény felfüggeszti az alkalmazás futását – kivéve az adott ablak kezelését – amíg az ablakot valamely elemével be nem zárja a felhasználó. A függvény segítségével az alkalmazás „megvárhatja” amíg a felhasználó az adott üzenet-ablakkal van elfoglalva.

E függvény hívása után általában a `gtk_widget_destroy()` függvényt hívjuk, hogy a dialógusablakot megsemmisítsük.

A függvény visszatérési értéke és argumentuma a következők:

visszatérési érték A függvény visszatérési értéke megadja, hogy az ablak milyen körülmények közt záródott be. A függvény a következő állandók egyikét adja vissza:

GTK_RESPONSE_DELETE_EVENT Az ablak nem nyomógomb hatására zárult be.

GTK_RESPONSE_OK A bezárás az „OK” feliratú nyomógomb segítségével történt.

GTK_RESPONSE_CANCEL Bezárás a „mégsem” nyomógommbal.

GTK_RESPONSE_CLOSE Bezárás a „bezár” nyomógommbal.

GTK_RESPONSE_YES Bezárás az „igen” nyomógommbal.

GTK_RESPONSE_NO Bezárás a „nem” nyomógommbal.

ablak A megjelenítendő ablakot jelöli a memóriában.

Az üzenet-ablakok kapcsán érdemes megemlítenünk, hogy az üzenet ablakokhoz további nyomógombok adhatók a `gtk_dialog_add_buttons()` függvény segítségével. Erről a függvényről a GTK+ dokumentációjában olvashatunk részletesebben.

30. példa. A következő függvény bemutatja, hogyan hozhatunk létre egyszerű üzenetablakot. Figyeljük meg, hogy az üzenet-ablak létrehozásához ki kellett derítenünk a szülőablak címét, és ezt a szokásos módon, a Glade által készített `lookup_widget()` függvény segítségével tettünk meg. Az is megfigyelhető a programrészletben, hogy az üzenet-ablakot a bezárás után `gtk_widget_destroy()` függvény segítségével megsemmisítettük, hogy ne foglaljon feleslegesen memóriát.

```

1 void
2 on_item_error_activate(GtkMenuItem *menuitem,
3                       gpointer      user_data)
4 {
5     GtkWidget *window;
6     GtkWidget *dialog;
7
8     window = lookup_widget(GTK_WIDGET(menuitem), "app1");
9     dialog = gtk_message_dialog_new (GTK_WINDOW(window),
10                                     GTK_DIALOG_DESTROY_WITH_PARENT,
11                                     GTK_MESSAGE_ERROR,
12                                     GTK_BUTTONS_CLOSE,
13                                     "A hibaüzenet szövege: '%s': %s",
14                                     "állománynév", g_strerror(errno));
15     gtk_dialog_run (GTK_DIALOG (dialog));
16     gtk_widget_destroy (dialog);
17 }

```

31. példa. A következő programrészletben már nem csak egy egyszerű üzenetet fogalmazunk meg, hanem egy kérdést teszünk fel a felhasználónak, ezért tudnunk kell, hogy a felhasználó melyik nyomógombot nyomta le az ablak bezárásakor. Ennek megfelelően a 16. sorban elmentjük a `gtk_dialog_run()` függvény visszatérési értékét és kiértékeljük 19–28. sorokban.

```

1 void
2 on_item_question_activate(GtkMenuItem *menuitem,
3                           gpointer      user_data)
4 {
5     GtkWidget *window;
6     GtkWidget *dialog;
7     gint response;
8
9     window = lookup_widget(menuitem, "app1");
10    dialog = gtk_message_dialog_new (GTK_WINDOW(window),
11                                    GTK_DIALOG_DESTROY_WITH_PARENT,
12                                    GTK_MESSAGE_QUESTION,
13                                    GTK_BUTTONS_YES_NO,
14                                    "A kérdés szövege...");
15    response = gtk_dialog_run (GTK_DIALOG (dialog));
16    gtk_widget_destroy (dialog);
17
18    switch (response) {

```



```

19     case GTK_RESPONSE_YES:
20         printf("Rendben.\n");
21         break;
22     case GTK_RESPONSE_NO:
23         printf("Nem?\n");
24         break;
25     default:
26         printf("Nincs válasz...\n");
27 }
28 }
```

7.2.2. Az állománynév-ablak

Sokszor előfordul, hogy az alkalmazásunk valamilyen állomány- vagy könyvtárnevet szeretne megtudakolni a felhasználótól. Ehhez a feladathoz egy igen kényelmes és könnyen kezelhető eszközt biztosít a GTK+ programkönyvtár az állománynév-ablakkal amelyet a 7.4. ábrán láthatunk.

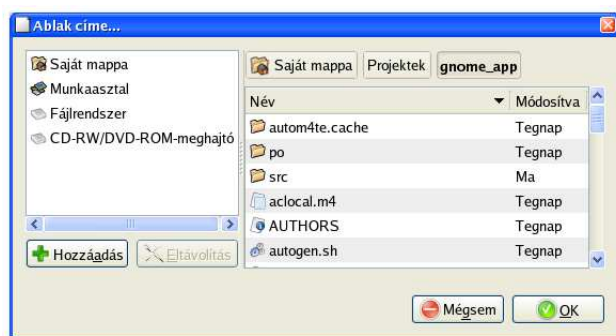
Az ábrán látható, hogy az állomány vagy könyvtár kiválasztásakor a felhasználó megnézheti, hogy milyen állományok és könyvtárak találhatók az egyes adathordozón, adathordozókat illeszthet be a könyvtárszerkezetbe és az ikonok segítségével könnyen felismerheti az egyes állományok típusát.

Az állománynév ablak egy gyorskereső szolgáltatással is fel van szerelve. Ha a felhasználó elkezd begépelni a keresett állománynevet, a bal oldalon található lista folyamatosan mutatja a találatokat. Ennek a szolgáltatásnak köszönhetően a felhasználónak nem kell végigböngésznie a listát, elegendő, ha a keresett név első néhány betűjét begépel.

Összességében elmondható, hogy az állománynév-ablak igen fejlett és kényelmes eszközt biztosít mind a programozó, mind pedig a felhasználó számára és mivel kevés olyan alkalmazás van, amelyik egyáltalán nem kezel állományokat, mindenképpen érdemes megismerkedni vele.

Az állománynév ablak kezelésére használhatók a következő függvények.

```
GtkWidget *gtk_file_chooser_dialog_new(const gchar
```



7.4. ábra. Az állománynév ablak


```
*címsor, GtkWindow *szülő, GtkFileChooserAction
jelleg, const gchar *első_gomb, GtkResponseType
első_válasz, ...);
```

A függvény állománynév-ablak létrehozására használható. A függvény a megadott értékeknek megfelelően létrehoz és megjelenít egy új ablakot, amelyben a felhasználó állomány- vagy könyvtárnevek közül választhat.

visszatérési érték A függvény visszatérési értéke az újonnan létrehozott ablakot jelöli a memóriában. Ez az ablak ugyanolyan módon használható a `gtk_dialog_run()` és a `gtk_widget_destroy()` függvényekkel, mint az üzenet-ablakok.

címsor A szöveg, amely az újonnan létrehozott ablak címezejében fog megjelenni.

szülő Az újonnan létrehozott ablak szülőablaka, melynek szerepe és használata megegyezik az üzenet-ablakoknál megismert szülő ablak szerepével és használatával.

jelleg Megadja, hogy milyen jellegű feladatot lásson el az állománynév-ablak. A jelleg meghatározza, hogy milyen módon viselkedjen az új ablak, például azzal, hogy meghatározza melyik állományokat tudja kiválasztani a felhasználó. Itt a következő állandók közül választhatunk:

GTK_FILE_CHOOSER_ACTION_OPEN Az állománynévre megnyitás céljából van szüksége az alkalmazásnak, a felhasználó tehát a létező állományok közül választhat.

GTK_FILE_CHOOSER_ACTION_SAVE Az állománynévre mentés céljából van szüksége az alkalmazásnak, a felhasználó tehát választhat nem létező állományt is.

GTK_FILE_CHOOSER_ACTION_SELECT_FOLDER Az alkalmazásnak könyvtárnévre van szüksége.

GTK_FILE_CHOOSER_ACTION_CREATE_FOLDER Az alkalmazás könyvtárat szeretne létrehozni, ezért csak nem létező állománynevek választhatók.

első_gomb Az ablak aljára helyezendő első nyomógomb azonosítója. A GTK+ alapértelmezés szerint nem helyez nyomógombokat az állománynév-ablak aljára, ezért legalább egy nyomógombról gondoskodnunk kell.

Itt igen sok féle azonosítót megadhatunk, de a legfontosabbak a következők:

GTK_STOCK_APPLY Elfogad.

GTK_STOCK_CANCEL Mégsem.
 GTK_STOCK_HELP Segítség.
 GTK_STOCK_NEW Új.
 GTK_STOCK_NO Nem.
 GTK_STOCK_OK Ok.
 GTK_STOCK_OPEN Megnyitás.
 GTK_STOCK_PRINT Nyomtatás.
 GTK_STOCK_REMOVE Törlés.
 GTK_STOCK_SAVE Mentés.
 GTK_STOCK_YES Igen.

A nyomógombok a felhasználó által beállított nyelven, a felhasználó által beállított témának megfelelő ikonnal jelennek meg.

első_válasz A nyomógombnak megfelelő válasz, amely megadja, hogy ha a felhasználó az adott nyomógombot lenyomja, a `gtk_dialog_run()` milyen értéket adjon vissza.

Itt a következő értékek közül választhatunk (hogya mit értünk az egyes állandókon, az viszont a belátásunkra van bízva, hiszen a mi programunk fogja kiértékelni a választ):

GTK_RESPONSE_OK Ok.
 GTK_RESPONSE_CANCEL Mégsem.
 GTK_RESPONSE_CLOSE Bezár.
 GTK_RESPONSE_YES Igen.
 GTK_RESPONSE_NO Nem.
 GTK_RESPONSE_APPLY Elfogad.
 GTK_RESPONSE_HELP Segítség.

... A további helyeken tetszőleges számú további nyomógombot és választ adhatunk meg. A nyomógombok az ablak alsó részén balról jobbra haladva jelennek meg.

Fontos viszont, hogy az utolsó nyomógomb-jelentés páros után egy `NULL` értéket kell megadnunk.

```
gboolean gtk_file_chooser_set_current_folder(GtkFileChooser
    *ablak, const gchar *könyvtárnév);
```

A függvény segítségével beállíthatjuk, hogy az állománynév-ablak melyik könyvtárat mutatja. A legtöbb esetben a `gtk_dialog_run()` hívása előtt használjuk ezt a függvényt, ha meg akarunk bizonyosodni arról, hogy az ablak egy adott könyvtárat mutat induláskor.

visszatérési érték Ha sikerült beállítani a könyvtárat `TRUE`, ha nem `FALSE`.

ablak Az állítani kívánt állománynév-ablakot jelöli a memóriában.

könyvtárnév A könyvtár neve.

```
gchar *gtk_file_chooser_get_filename(GtkFileChooser
    *ablak);
```

A függvény segítségével megtudhatjuk, hogy mi a teljes neve annak a könyvtárnak vagy állománynak, ami éppen ki van választva. Ezt a függvényt általában a `gtk_dialog_run()` függvény után hívjuk, amikor a felhasználó már kiválasztotta az állománynevet és lenyomta valamelyik nyomógombot.

ablak A lekérdezendő állománynév-ablakot jelöli a memóriában.

visszatérési érték A kiválasztott állomány teljes elérési útja vagy `NULL`, ha a felhasználó nem jelölt ki állománynevet.

Ha a visszatérési érték nem `NULL`, akkor a `g_free()` függvény segítségével fel kell szabadítanunk, ha már nincs szükségünk rá.

Az állománynév-ablak még sok más hasznos eszközt is tartalmaz. Ezekről bővebben a GTK+ programkönyvtár dokumentációjában olvashatunk.

A következő példa bemutatja hogyan készíthetünk és használhatunk egyszerűen állománynév-ablakot.

32. példa. A következő programrészlet bemutatja hogyan jeleníthetünk meg állománynév kérésére használható ablakot és hogyan használhatjuk fel a felhasználó által választott állománynevet.

```
1 void
2 on_item_filename_activate(GtkMenuItem *menuitem,
3                             gpointer     user_data)
4 {
5     GtkWidget *window;
6     GtkWidget* dialog;
7     gchar *filename;
8
9     window = lookup_widget(GTK_WIDGET(menuitem),
10                             "window2");
11     dialog = gtk_file_chooser_dialog_new("Ablak címe...",
12                                         GTK_WINDOW(window),
13                                         GTK_FILE_CHOOSER_ACTION_OPEN,
14                                         GTK_STOCK_CANCEL, GTK_RESPONSE_CANCEL,
15                                         GTK_STOCK_OK, GTK_RESPONSE_OK,
16                                         NULL);
```

```

17
18     if (gtk_dialog_run(GTK_DIALOG(dialog)) ==
19         GTK_RESPONSE_OK){
20         filename = gtk_file_chooser_get_filename(
21             GTK_FILE_CHOOSER(dialog));
22         printf("A választott állomány: '%s'\n", filename);
23     }else{
24         printf("Nincs választott állománynév.\n");
25     }
26
27     if (filename != NULL)
28         g_free (filename);
29
30     gtk_widget_destroy(dialog);
31 }

```

7.2.3. A betűtípus-ablak

Grafikus alkalmazást készítve szükségünk lehet olyan eszközre, amely lehetővé teszi a felhasználó számára, hogy betűtípust válasszon. A GTK+ könyvtár erre a célra egy könnyen kezelhető, egyszerű ablakot, a betűtípus ablakot biztosítja. Ez az ablak igen egyszerűen és kényelmesen létrehozható a megfelelő függvények hívásával.

A 7.5. ábra a betűtípus-ablakot mutatja be. Jól látható, hogy a kezelése egyszerű és kényelmes. Megfigyelhetjük, hogy egy előnézeti képen be is mutatja, hogy az adott betűtípus milyen írásképet ad a képernyőn. Az előnézetben látható szöveg szabadon változtatható. Alapértelmezés szerint ez a szöveg az „árvíztűrő tükörfúrógép”, amelynek sok értelme ugyan nincs, de tartalmazza a magyar nyelvre jellemző ékezetes karaktereket. Ezt a szöveget a megfelelő függvény segítségével tetszőleges szöveges értékre megváltoztathatjuk.

A betűtípus-ablak létrehozásához és a használatához a következő függvényekre van szükségünk:



7.5. ábra. A betűtípus-ablak

```

GtkWidget *gtk_font_selection_dialog_new(const gchar
    *cím);

```

A függvény segítségével új, betűtípus kiválasztására szolgáló ablakot hozhatunk létre. A függvénynek paraméterként átadott cím a létrehozott ablak címmezejében megjelenő szöveget adja meg.

```
void gtk_font_selection_dialog_set_preview_text(GtkFont-
SelectionDialog *ablak, const gchar *szöveg);
```

A függvény segítségével a betűtípus-ablakban a választott betű tulajdonságainak bemutatására szolgáló szöveget állíthatjuk be. A függvény első paramétere az ablakot jelöli, a második pedig a szöveget, ami az ablakban meg fog jelenni.

```
gchar *gtk_font_selection_dialog_get_font_name(GtkFont-
SelectionDialog *ablak);
```

A függvény segítségével megállapíthatjuk, hogy a felhasználó milyen betűtípust választott. A függvény paramétere a betűtípus-ablakot jelöli a memóriában, míg a visszatérési értéke megadja a választott betűtípus nevét.

A függvény által visszaadott mutató egy memóriaterületet jelöl a memóriában, amelyet nekünk kell felszabadítanunk, ha már nincs rá szükségünk.

33. példa. A következő példaprogram bemutatja hogyan hozhatunk létre betűtípust kérő ablakot és hogyan használhatjuk fel a felhasználó által kiválasztott betűtípus nevét.

```
1 void
2 on_new_font_window_activate(GtkMenuItem *menuitem,
3                             gpointer      user_data)
4 {
5     GtkWidget *dialog;
6     gchar *fontname;
7
8     dialog = gtk_font_selection_dialog_new("Betűtípusok");
9     if (gtk_dialog_run (GTK_DIALOG (dialog)) !=
10         GTK_RESPONSE_OK){
11         gtk_widget_destroy(dialog);
12         return;
13     }
14
15     fontname = gtk_font_selection_dialog_get_font_name(
16         GTK_FONT_SELECTION_DIALOG(dialog));
17     printf("A választott betűtípus: '%s'\n", fontname);
18
19     g_free(fontname);
```

```
20     gtk_widget_destroy (dialog);
21 }
```

A könyvben nem térünk ki részletesen arra, hogy hogyan használhatjuk a betűtípus választó ablakban kiválasztott betűtípus nevét, de egy példaprogramon keresztül bemutatjuk hogyan állíthatjuk be vele egy tetszőleges képernyőelem alapértelmezett betűtípusát.

34. példa. A következő sorok egy visszahívott függvényt mutatnak be, amely egy betűtípus-ablakban lehetőséget ad a felhasználónak arra, hogy egy képernyőelem alapértelmezett betűtípusát megváltoztassa.

```
1  void
2  on_textview_window_button_font_clicked(
3      GtkWidget *button,
4      gpointer  user_data)
5  {
6      GtkWidget *text_view;
7      GtkWidget *dialog;
8      gchar *fontname;
9      PangoFontDescription *font_desc;
10
11     text_view = lookup_widget(GTK_WIDGET(button),
12                             "textview");
13     /*
14      * A jelenlegi betűtípus lekérdezése és a betűtípus
15      * választó ablak létrehozása.
16      */
17     fontname = pango_font_description_to_string(
18         text_view->style->font_desc);
19     dialog = gtk_font_selection_dialog_new(
20         "Betűtípus kiválasztása");
21     gtk_font_selection_dialog_set_font_name(
22         GTK_FONT_SELECTION_DIALOG(dialog), fontname);
23     g_free(fontname);
24
25     /*
26      * A betűtípus választó ablak megjelenítése.
27      */
28     if (gtk_dialog_run(GTK_DIALOG(dialog)) !=
29         GTK_RESPONSE_OK){
30         gtk_widget_destroy(dialog);
31         return;
32     }
```

```

33
34  /*
35   * A kiválasztott betűtípus érvényesítése.
36   */
37  fontname = gtk_font_selection_dialog_get_font_name(
38      GTK_FONT_SELECTION_DIALOG(dialog));
39  font_desc = pango_font_description_from_string(
40      fontname);
41  gtk_widget_modify_font(text_view, font_desc);
42
43  /*
44   * Az erőforrások felszabadítása.
45   */
46  g_free(fontname);
47  gtk_widget_destroy(dialog);
48  }

```

FIXME: ezt meg kell írni részletesre, mert máshol nem mutatjuk be ezeket az eszközöket...

7.2.4. A színválasztó-ablak

Szintén igen egyszerűen létrehozható és nagyon hasznos a színválasztó-ablak, amely a 7.6. ábrán látható.



7.6. ábra. A színválasztó-ablak

rában.)

Roppant hasznos eszköz a kép alsó részén látható, pipettát formázó ikon, amelynek segítségével a képernyő bármely pontjának színét beállíthatjuk a színháromszögön. Az ikonra kattintva egy pipetta alakú

Az ablakban található színháromszög segítségével tetszőleges színt kiválaszthatunk. A háromszöget körbeforgathatjuk a kerék segítségével és a háromszög bármely pontjára kattintva kiválaszthatunk egy tetszőleges színt. Az oldalt látható *szín neve* mezőbe színek angol neveit vagy a három alapszín kódját írhatjuk. (Színek neveit GNU/Linux rendszereken a színnadatbázis tartalmazza, amelyet általában az *rgb.txt* állományban találunk a grafikus kiszolgáló valamelyik könyvtá-

egérkurzort kapunk, amelyek bármely alkalmazás bármely pontjára kattintva elmenthetjük annak színét. Ha valamelyik program vagy kép színe megtetszik azt azonnal be is állíthatjuk a saját alkalmazásunk részére.

A GTK+ programkönyvtár a színek kezelésére a `GdkColor` típust használja, amelyet a GDK programkönyvtár valósít meg. A `GdkColor` típust számtalan függvény hívásakor használhatjuk a színek kezelésére, az egyszerű képernyőelemek esetében azonban általában nincs szükségünk a színek használatára.

A színválasztó ablak kezelését kissé komplikálttá teszi a használt sokféle típus. A színválasztó ablak létrehozásakor egy `GtkWidget` típust kapunk, amely valójában egy párbeszédablak címe a memóriában. A párbeszédablak típusa tulajdonképpen `GtkColorSelectionDialog`, ami rendelkezik egy `colorsel` nevű, `GtkColorSelection` mezővel. Ezt a mezőt tudjuk a színválasztó ablak kezelésére használni. A függvények után bemutatott példaprogram remélhetőleg érthetőbbé teszi ezt a típuskavalkádot.

A színválasztó-ablak létrehozásakor és használatakor a következő függvények a legfontosabbak:

```
gint gdk_color_parse(const gchar *spec, GdkColor *szín);
```

A függvény segítségével a grafikus felhasználói felület nyilvántartásából kereshetünk ki neve alapján egy színt. A függvény kikeresi a színt és a `GdkColor` típusú adatszerkezetet a megfelelően kitöltve elérhetővé teszi a számunkra. A `GdkColor` adatszerkezet rendelkezik három `uint16` (előjel nélküli 16 bites egész) mezővel `red`, `green` és `blue` néven, amelyek rendre a vörös, zöld és kék színek komponenseket hordozzák.

A függvény első paramétere a szín neve, amelyet a színadatbázisban keresünk.

A függvény második paramétere azt a memóriaterületet jelöli, ahová a kikeresett színt gépi formában elhelyezi a függvény. A `gdk_color_parse()` erre a memóriaterületre ír, a mutatónak tehát egy `GdkColor` típusú adatszerkezet hordozására alkalmas memóriaterületet kell jelölnie.

A függvény visszatérési értéke igaz, ha a színt sikeresen megtalálta az adatbázisban, illetve hamis, ha a szín nem található.

```
GtkWidget *gtk_color_selection_dialog_new(const gchar *cím);
```

A függvény segítségével új színválasztó ablakot hozhatunk létre. A függvénynek átadott paraméter a színválasztó ablak címsorában megjelenő szöveget határozza meg.

```
void gtk_color_selection_set_has_opacity_control(GtkColorSelection *választó, gboolean átlátszóság);
```

A függvény segítségével beállíthatjuk, hogy a színválasztó ablakban

jelenjen-e meg az átlátszóság beállítására szolgáló eszköz. A függvény első paramétere a színválasztó ablak színválasztó része (`colorsel` komponense), a második paramétere pedig egy logikai érték, amely meghatározza, hogy megjelenjen-e az átlátszóság az ablakban.

`void gtk_color_selection_set_current_color(GtkColorSelection *választó, const GdkColor *szín);` A függvény segítségével beállíthatjuk a színválasztó ablakban látható színt. A függvény első paramétere a színválasztó ablak színválasztó része (`colorsel` komponense), második paramétere pedig a megjelenítendő színt jelöli a memóriában.

`void gtk_color_selection_set_current_alpha(GtkColorSelection *választó, guint16 alpha);` A függvény segítségével beállíthatjuk a színválasztó ablakban megjelenő átlátszóságot (valójában átlátszatlanságot).

A függvény első paramétere a színválasztó ablak színválasztó része (`colorsel` komponense), második paramétere pedig az átlátszóságot jelölő szám. Ez utóbbi 0 (teljesen átlátszó) és 65535 (teljesen átlátszatlan) érték közti egész szám lehet.

`void gtk_color_selection_get_current_color(GtkColorSelection *választó, GdkColor *szín);` A függvény segítségével a színválasztó ablakban beállított színt kérdezhetjük le. A függvény első paramétere a színválasztó ablak színválasztó része, a második paramétere pedig azt a memóriaterületet jelöli a memóriában, ahova a függvény a beállított színt el fogja helyezni.

`guint16 gtk_color_selection_get_current_alpha(GtkColorSelection *választó);` A függvény segítségével lekérdezhetjük a színválasztó ablakban beállított átlátszóságot. A függvény paramétere a színválasztó ablak színválasztó része, a visszatérési értéke pedig a beállított átlátszóság-érték.

A színválasztó ablak létrehozását, beállítását és a felhasználó által beállított értékek lekérdezését mutatja be a következő példaprogram.

35. példa. A következő programrészlet szín és átlátszóságérték beállítására alkalmas ablakot helyez el a képernyőn, majd a felhasználó által beállított értékeket kiírja a szabványos kimenetre. A program az ablakot kezdetben teljesen átlátszatlan zöld színre állítja. A zöld szín a grafikus felhasználoi felület adatbázisából származik.

```
1 void
2 on_new_color_window_activate(GtkMenuItem *menuitem,
```

```

3          gpointer      user_data)
4  {
5      GtkWidget *dialog;
6      GdkColor color;
7      GtkColorSelectionDialog *color_sel_dialog;
8      guint16 alpha;
9
10     /*
11      * Kikeressük a zöld színt (biztosan megtalálható).
12      */
13     gdk_color_parse("Green", &color);
14
15     /*
16      * Létrehozzuk a párbeszédablakot és az egyszerűbb
17      * használat érdekében két különböző típusú
18      * változóban is elhelyezzük a mutatóját.
19      */
20     dialog = gtk_color_selection_dialog_new("Színek");
21     color_sel_dialog = GTK_COLOR_SELECTION_DIALOG(dialog);
22
23     /*
24      * A párbeszédablak alapbeállítása.
25      */
26     gtk_color_selection_set_has_opacity_control(
27         GTK_COLOR_SELECTION(color_sel_dialog->colorsel),
28         TRUE);
29     gtk_color_selection_set_current_color(
30         GTK_COLOR_SELECTION(color_sel_dialog->colorsel),
31         &color);
32     gtk_color_selection_set_current_alpha(
33         GTK_COLOR_SELECTION(color_sel_dialog->colorsel),
34         65535);
35
36     /*
37      * Ha a felhasználó az OK gombot nyomta meg kiírjuk a
38      * beállított értékeket, ha nem, egyszerűen
39      * eltüntetjük az ablakot.
40      */
41     if (gtk_dialog_run(GTK_DIALOG(dialog)) ==
42         GTK_RESPONSE_OK){
43         gtk_color_selection_get_current_color(
44             GTK_COLOR_SELECTION(color_sel_dialog->colorsel),
45             &color);
46         alpha = gtk_color_selection_get_current_alpha(

```

198

```

47         GTK_COLOR_SELECTION(
48             color_sel_dialog->color_sel));
49     printf("A szín: \n"
50           "  vörös = %u\n"
51           "  zöld  = %u\n"
52           "  kék   = %u\n"
53           "Átlátszóság: %u\n",
54           color.red,
55           color.green,
56           color.blue,
57           alpha);
58 }
59 gtk_widget_destroy(dialog);
60 }
```

A program könnyen megérthető, ha figyelmesen áttanulmányozzuk és megfigyeljük a függvényben található változók típusát.

Nem tárgyaljuk részletesen, hogy hogyan lehet beállítani az egyes képernyőelemek színeit, de a következő példában bemutatunk egy függvényt, amely közvetlenül módosítja egy képernyőelem megjelenését a háttérszín beállításával.

36. példa. A következő függvény megjelenít egy színválasztó-ablakot, beállítja az ablakban a szövegszerkesztő képernyőelem hátterének színét és megváltoztatja a szövegszerkesztő hátterének színét, ha a felhasználó új színt választ ki.

```

1  void
2  on_textview_window_button_color_clicked(
3      GtkButton *button,
4      gpointer  user_data)
5  {
6      GtkWidget *text_view;
7      GtkWidget *dialog;
8      GdkColor color;
9      GtkColorSelectionDialog *color_sel_dialog;
10
11     text_view = lookup_widget(GTK_WIDGET (button),
12                               "textview");
13     dialog = gtk_color_selection_dialog_new("Háttérszín");
14     color_sel_dialog = GTK_COLOR_SELECTION_DIALOG(dialog);
15
16     /*
17     * A párbeszédablak alapbeállítása.
```

```

18      */
19      gtk_color_selection_set_current_color(
20          GTK_COLOR_SELECTION(color_sel_dialog->color_sel),
21          &text_view->style->base[0]);
22
23      /*
24       * A dialógusablak futtatása és a szín beállítása.
25       */
26      if (gtk_dialog_run(GTK_DIALOG(dialog)) ==
27          GTK_RESPONSE_OK){
28          gtk_color_selection_get_current_color(
29              GTK_COLOR_SELECTION(color_sel_dialog->color_sel),
30              &color);
31          // Előtér és háttérszínek.
32          //gtk_widget_modify_fg(text_view, GTK_STATE_NORMAL,
33          //    &color);
34          //gtk_widget_modify_bg(text_view, GTK_STATE_NORMAL,
35          //    &color);
36          // Szöveg karaktereinek színe.
37          //gtk_widget_modify_text(text_view,
38          //    GTK_STATE_NORMAL, &color);
39
40          // A szöveg hátterének színe.
41          gtk_widget_modify_base(text_view, GTK_STATE_NORMAL,
42              &color);
43      }
44      gtk_widget_destroy(dialog);
45  }

```

200

8. fejezet

Összetett képernyőelemek

Ebben a fejezetekben azokról a képernyőelemekről lesz szó, amelyek felépítésükben vagy használatukban összetettek, bonyolultak. Az ilyen képernyőelemeket nem egyszer bonyolult függvények, összetett programok kezelik, ezért az ilyen képernyőelemek nem egyszer komoly programozói tudást tesznek szükségessé.

8.1. Az alkalmazás-ablak

Ha a Glade segítségével GNOME alkalmazást készítünk, a *paletta* ablakból választhatjuk a *gnome alkalmazás ablak* eszközt, amely összetett, több képernyőelemet tartalmazó ablak létrehozására használható. Ezt az ablaktípust mutatja be a 8.1. ábra.

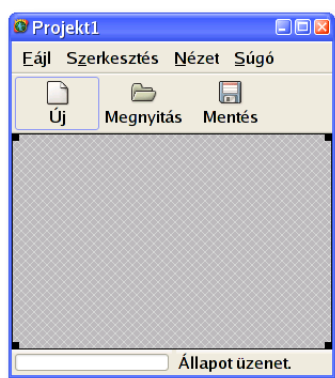
A GNOME alkalmazás ablak felső részében egy menüsor foglal helyet. Ebben a menüsorban a Glade a szokásos módon menüket és menüpontokat helyez, de a létrehozás után lehetőségünk nyílik a menü módosítására is. Ehhez ki kell jelölnünk a menüsávot, majd a *tulajdonságok* ablakban a *menük szerkesztése* nyomógombra kell kattintanunk. A menük szerkesztéséről bővebben az 52. oldalon található 3.1.10. szakaszban olvashattunk.

8.1.1. Az elmozdítható eszközsáv

A menüsor alatt egy eszközsávot látunk. A Glade az alkalmazás-ablak eszközsávjában a szokásos elemeket elhelyezi, de bármelyik nyomógombot törölhetjük és új nyomógombokat is létrehozhatunk. Az eszközsávon belül használatos nyomógombokból többféle is a rendelkezésünkre áll. Egyszerű nyomógombok, kapcsoló jellegű „bennmaradó” nyomógombok

202

és rádiógomb jellegű nyomógombok is elérhetők külön, az eszközsáv számára. A *paletta* ablakban még elválasztó elemeket is találunk, amelyek az eszközsáv nyomógombjainak csoportosítására használhatók. Az eszközsáv – a megfelelő módon elhelyezve – elmozdítható, az alkalmazás területén belül vízszintesen és függőlegesen is megjelenhet, ezért „elmozdítható” eszközsávról beszélhetünk.



8.1. ábra. A GNOME alkalmazás ablak

A GTK programkönyvtár az eszközsáv kezelésére a *GtkToolbar* típust biztosítja a programozó számára. A *GtkToolbar* meglehetősen egyszerű képernyőelem, a programozónak nem sok munkát jelent a használata. Általában elegendő, ha az eszközsáv beállításait a Glade *tulajdonságok* ablakában a program megírásakor elvégezzük, a programunkban pedig kizárólag az eszközsávban található nyomógombok visszahívott függvényeivel foglalkozunk.

A Glade *tulajdonságok* ablakában az eszközsáv beállításai közül felülről az első említésre méltó a *keret szélessége* mező, ahol beállíthatjuk, hogy az eszközsáv körül hány képpontnyi üres helyet akarunk kihagyni. Ennek a mezőnek az alapértelmezett értéke 0, ami tulajdonképpen meg is felel a legtöbb alkalmazás számára. Más dobozok körül szokás néhány képpontnyi helyet kihagyni, az eszközsáv esetében azonban ez a szegély nem volna esztétikus. A

következő fontos tulajdonság a *méret*, ami meghatározza, hogy az eszközsávban hány képernyőelemnek tartunk fel helyet. Ez a tulajdonság is hasonlóképpen állítható az összes doboz jellegű képernyőelem esetében, a különbség talán csak annyi, hogy az eszközsáv esetében ezt a tulajdonságot rendszeresen változtatjuk is, hiszen általában elég sokáig tart, amíg a programozó eldönti, hogy mit is akar elhelyezni az eszközsávban. Szerencsére a Glade elég rugalmasan kezeli az eszközsáv méretét, a benne megjelenő elemek sorrendjét pedig könnyedén megváltoztathatjuk a vágólap segítségével, amelyet a Glade szerkesztőablakában a jobb egérgombbal érhetünk el.

A *tulajdonságok* ablak következő mezője az *elhelyezkedés* feliratot viseli. Itt beállíthatjuk, hogy az eszközsáv vízszintesen vagy függőlegesen jelenjen-e meg a képernyőn. A legtöbb esetben a vízszintes megjelenést választjuk, amit aztán a felhasználó a program futása közben megváltoztathat. A *stílus* mezőben beállíthatjuk, hogy az eszközsáv nyomógombjai hogyan jelenjenek meg. Választhatjuk az ikonszerű és a szövegszerű megjelenést, valamint, ha az utóbbit választva meghatározhatjuk, hogy a nyomógombok címkéi az ikonok mellett, vagy az ikonok alatt jelenjen-e meg. A nyomógombok megjelenését szintén megváltoztathatja a felhasználó, a program használata során.

A *segédszöveg* kapcsoló segítségével beállíthatjuk, hogy a nyomógombokhoz tartozó segédszövegek – közismert nevükön a *tippek* – megjelenjenek-e az eszközsávban. Ezt a kapcsolót általában bekapcsolva tartjuk, hiszen a segédszövegek kikapcsolása nem sok előnnyel kecsegtet.

A *nyíl megjelenítése* (*show arrow*) kapcsolóval meghatározhatjuk, hogy mi történjék, ha az eszközsáv nem fér el az alkalmazás ablakában. Ha a *nyíl megjelenítése* menüpontot kiválasztjuk, a csonkolt eszközsáv jobb szélén egy nyíl alakú szimbólum jelenik meg, amelynek segítségével a felhasználó az éppen nem látható nyomógombok funkcióit is elérheti. Nyilvánvaló, hogy ezt a szolgáltatást sem érdemes kikapcsolnunk, ha csak valamilyen különleges célunk nincs ezzel.

Az eszközsávban általában nyomógombokat helyezünk el. A GTK+ programkönyvtár különféle, kimondottan az eszközsáv számára készített nyomógombokat biztosít erre a célra. Természetesen tehetünk egyszerű nyomógombokat is az eszközsávba, de néhány tulajdonságuk miatt az eszközsávba szánt nyomógombtípusok kényelmesebbek.

Az eszközsáv számára készített egyszerű nyomógomb típusa a *GtkToolButton*, ami leginkább a *GtkButton* típusú egyszerű nyomógombra hasonlít. A kapcsológomb eszközsávba szánt változata a *GtkToggleToolButton*, ami a *GtkToolButton* leszármazottja, attól csak abban különbözik, hogy ha lenyomjuk „bennmarad”. Jó példa az eszközsáv kapcsológombjának alkalmazására a különféle kiadványszerkesztő programok eszközsávjainak *aláhúzott*, *dőlt* és *félkövér* nyomógombjai, amelyek a szöveg betűváltozatának beállítására használhatók.

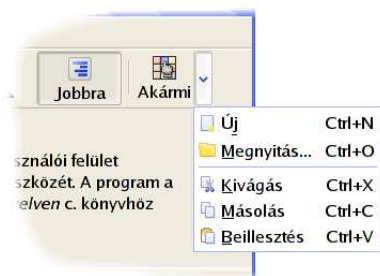
Az eszközsáv számára tervezett rádiógomb, a *GtkRadioToolButton* ami a *GtkToggleToolButton* leszármazottjaként, attól csak abban különbözik, hogy az egyazon csoportba tartozó rádiógombok egymást kikapcsolják. Jó példa az alkalmazásukra a kiadványszerkesztő program *jobbra*, *balra* és *középre* rendező nyomógombjai.

Ritkábban használatos, de azért hasznos a menük megjelenítésére is használható *GtkMenuToolButton*, amit a 8.2. ábrán láthatunk. Az ilyen nyomógombok mellett egy nyíl is megjelenik, amit a felhasználó arra használhat, hogy egy menüt jelenítsen meg. A *GtkMenuToolButton* a *GtkToolButton* leszármazottja, és így a szokásos módon is használható, ha nem a nyílra, hanem magára a nyomógombra kattintunk.

Az eszközsáv elemeinek csoportosítására szolgál használható a *GtkSeparatorToolItem*, amit a 8.2. ábra két nyomógombja közt is megfigyelhetünk. Az ilyen elválasztóelemek segítségével az eszközsáv elemet jobbra rendezhetjük, ha magának az elválasztó elemnek a rajzolását kikapcsoljuk, a nyújtását viszont engedélyezzük.

A Glade *tulajdonságok* ablakában az eszközsáv nyomógombjainak minden típusánál beállíthatjuk a *sablon gomb* mezőben, hogy a nyomógomb melyik előre elkészített sablon alapján készüljön el. Ez a mező a szokásos

módon határozza meg a nyomógombon belül megjelenő ikon formáját és a nyomógomb címkéjét, figyelembe véve a beállított stílust és nyelvet.



8.2. ábra. A menü-gomb

Ha nem állítjuk be a nyomógomb sablont, akkor *címke* és az *ikon* mezőkben be kell állítanunk a nyomógombban megjelenő szöveget, illetve képet. Fontos tudnunk, hogy esetleg akkor is érdemes beállítanunk mind az *ikon*, mind pedig a *címke* mezők értékét, ha a Glade szerkesztőablakában valamelyik nem jelenne meg, hiszen az alkalmazás futtatása közben az eszközsáv megjelenése esetleg megváltozhat.

Az eszközsáv nyomógombjainak tulajdonságai közt beállíthatjuk, hogy az adott nyomógomb fontos-e. Ha a nyomógomb fontos és a felhasználó beállításai alapján a nyomógombok ikonjai alatt vagy mellett meg kell jelení-

teni a címkéket is, akkor a GTK+ programkönyvtár a címkéket is elhelyezi a nyomógombon belül. Úgy is mondhatnánk, hogy a címkék kizárólag csak a fontos nyomógombokban jelennek meg.

Szintén az eszköztár összes nyomógombja rendelkezik a *vízszintesen megjelenik* és a *függőlegesen megjelenik* tulajdonságokkal, amelyeket ki- és bekapcsolva meghatározzuk, hogy a nyomógomb megjelenjen-e amikor az eszközsáv vízszintes, illetve függőleges helyzetben van.

Az eszközsáv nyomógombjai közül a kapcsológombok és a rádiógombok esetében is megtalálható az *alapértelmezés szerint benyomva*, ami meghatározza, hogy a nyomógomb kezdetben lenyomva jelenjen-e meg.

A rádiógombok esetében megjelenik egy új és meglehetősen fontos tulajdonság, amelyet a *tulajdonságok* ablak *csoport* mezőjében adhatunk meg. A *csoport* mezőbe írt név segítségével meghatározhatjuk, hogy melyek azok a rádiógombok, amelyek kikapcsolják egymást. Az egy csoportba tartozó rádiógombok közül mindig csak egy lehet bekapcsolva.

Az eszközsáv nyomógombjai ugyanazokat az üzeneteket küldik, mint amelyeket az egyszerű nyomógombok esetében megszoktunk. Ezek közül a legfontosabbak a következők:

clicked Ezt az üzenetet az eszközsáv minden nyomógombja képes küldeni, amikor a felhasználó a nyomógombot aktiválta a billentyűzetrel vagy az egérrel.

toggled A kapcsológombok és a rádiógombok küldik ezt az üzenetet, amikor a felhasználó átkapcsolta őket. Érdemes megemlítenünk, hogy a rádiógombok akkor is küldik ezt az üzenetet, ha a felhasználó a csoport egy másik rádiógombjával kikapcsolja őket, azaz amikor a felhasználó egy rádiógombra kattintva átkapcsolja annak állapotát, akkor két rádiógomb is küldi a **toggled** üzenetet.

`show_menu` A menüvel ellátott nyomógomb küldi ezt az üzenetet, amikor a felhasználó a mellette található nyílra kattintva megjeleníti a nyomógombhoz tartozó menüt.

Az eszközsáv nyomógombjai érdekes módon nem a `GtkButton` leszármazottai, így hasonlítanak ugyan a szokásos nyomógombokra, de a nyomógombok kezelésére szolgáló függvények nem használhatók az esetükben. Az eszközsáv függvényeinek kezelésére a következő fontosabb függvényeket használhatjuk.

`void gtk_tool_button_set_label(GtkToolButton *nyomógomb, const gchar *címke);` A függvény segítségével a nyomógombban megjelenő címke szövegét megváltoztathatjuk. Ha a függvény második paraméterének értéke `NULL`, a GTK+ programkönyvtár nem jelenít meg címkét a nyomógombban belül.

`void gtk_tool_button_set_icon_widget(GtkToolButton *nyomógomb, GtkWidget *ikon);` A függvény segítségével megváltoztathatjuk a nyomógombon belül megjelenő ikont. A nyomógombban ikonként a második paraméter által kijelölt képernyőelem jelenik meg. A képek létrehozásáról a 37. oldalon kezdődő 3.1.2. szakaszban olvashatunk részletesebben.

`void gtk_toggle_tool_button_set_active(GtkToggleToolButton *nyomógomb, gboolean lenyomva);` A függvény segítségével beállíthatjuk, hogy az eszközsávban megjelenő kapcsológombok és rádiógombok be legyenek-e nyomva. Ha a második paraméter értéke `TRUE`, a nyomógomb bekapcsolt, ha viszont `FALSE`, kikapcsolt állapotba kerül.

`gboolean gtk_toggle_tool_button_get_active(GtkToggleToolButton *nyomógomb);` A függvény segítségével lekérdezhethetjük, hogy az eszközsávban található kapcsoló- illetve rádiógombok benyomott állapotban vannak-e. A függvény visszatérési értéke `TRUE`, ha a nyomógomb benyomott állapotban van, `FALSE`, ha nem.

`void gtk_menu_tool_button_set_menu(GtkMenuToolButton *nyomógomb, GtkWidget *menü);` A függvény segítségével a menüvel szerelt nyomógombhoz rendelhetjük hozzá a menüt.

A következő példa bemutatja a menüvel felszerelt nyomógomb használatát.

37. példa. *A következő sorok bemutatják hogyan rendelhetünk menüt az eszközsáv nyomógombjához.*

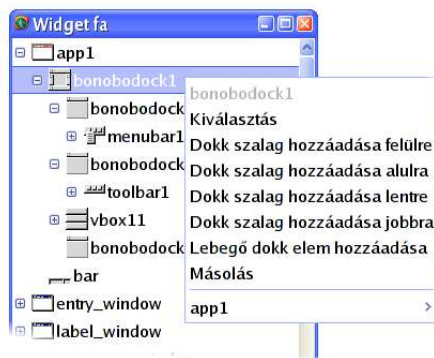
```

1 void
2 on_menutoolbutton1_realize(
3     GtkWidget *widget,
4     gpointer   user_data)
5 {
6     GtkWidget *menu;
7     menu = create_popup_menu1();
8     gtk_menu_tool_button_set_menu(
9         GTK_MENU_TOOL_BUTTON(widget),
10        menu);
11 }

```

A példaprogramban látható függvényt a GTK+ programkönyvtár akkor hívja, amikor a menüvel szerelt nyomógombot létrehozta. A függvény a 7. sorban hívja a Glade által létrehozott függvényt, hogy létrehozza a felbukkanó menüt, amelyet a Glade menüszerkesztőjében készítettünk.

A program ezek után a 8–10. sorokban a nyomógombhoz rendeli a felbukkanó menüt, ami ezután automatikusan megjelenik, ha a felhasználó a nyomógomb melletti nyílra kattint.



8.3. ábra. Új dokk szalag létrehozása

ban ki kell választanunk a **BonoboDock** elemet és az egér jobb gombjával megjelenített felbukkanó menü segítségével új **BonoboDockItem** képernyőelemet kell létrehoznunk. Az új képernyőelemben ezután a szokásos módon helyezhetünk el egy új eszközsávot (**GtkMenuBar**), az eszközsávon belül pedig a nyomógombokat.

Előfordulhat, hogy az elmozdítható eszközsávot ideiglenesen el akarjuk

Az elmozdítható eszközsávok készítéséhez a Glade a Bonobo programkönyvtár által létrehozott és kezelt **BonoboDock** típust használja. A **BonoboDock** doboz jellegű képernyőelem, ami több elmozdítható **BonoboDockItem** képernyőelem hordozására alkalmas. Ha figyelmesen megvizsgáljuk a Glade által létrehozott alkalmazás-ablak felépítését, akkor láthatjuk, hogy az eszközsáv azért mozdítható el, mert egy mozgatható **BonoboDockItem** képernyőelemen belül van.

Ha az alkalmazásunk számára új elmozdítható eszközsávot akarunk létrehozni, akkor a Glade **widget fa** ablaká-

rejtteni. Ekkor nem magát az eszközsávot kell elrejttenünk – hiszen akkor az üres szalag megmarad a képernyőn – hanem a **BonoboDockItem** képernyőelemet. Ezt mutatja be a következő példaprogram.

38. példa. A következő példaprogram bemutatja hogyan rejthetjük el az elmozdítható eszközsávot.

```

1 void
2 on_show_software_project_toolbar_activate(
3     GtkMenuItem *menuitem,
4     gpointer user_data)
5 {
6     GtkWidget *toolbar;
7     GtkCheckMenuItem *item;
8     gboolean visible;
9
10    item = GTK_CHECK_MENU_ITEM(menuitem);
11    visible = gtk_check_menu_item_get_active(item);
12    toolbar = lookup_widget(GTK_WIDGET(menuitem), "bar");
13
14    if (visible)
15        gtk_widget_show(toolbar->parent);
16    else
17        gtk_widget_hide(toolbar->parent);
18 }

```

A függvény egy kapcsolót tartalmazó menüpont üzenetkezelő függvénye, amelyet a GTK+ programkönyvtár akkor hív, ha a felhasználó aktiválta a menüpontot.

A függvény a 10–11. sorokban lekérdezi, hogy a menüpont bekapcsolt, vagy kikapcsolt állapotban van-e, majd a 12. sorban kideríti az eszközsáv memóriabeli címét.

A függvény ezek után a menüpont állapotától függően megjeleníti vagy elrejtí az eszközsáv képernyőelem képernyőbeli szülőjét, azt a képernyőelemet, amely az eszközsávot magába foglalja. Figyeljük meg a 15. és 17. sorokban, ahogyan a **GtkWidget** adatszerkezetben található **parent** mező segítségével megjelenítjük, illetve elrejtjük a szülő képernyőelemet!

8.1.2. A fő képernyőelem

Az eszközsáv alatt egy üres terület található. Itt helyezhetjük el azokat a képernyőelemeket, amelyeket ebben a főablakban kívánunk megjeleníteni. Ha például szövegszerkesztő programot akarunk csinálni, akkor egy szövegszerkesztő képernyőelemet helyezhetünk el itt, ha valamilyen

208

egyszerű alkalmazást tervezünk, akkor pedig néhány egyszerű képernyőelemből készíthetünk összetett grafikus felhasználói felületet ezen a helyen.

8.1.3. Az állapotosor

A fő képernyőelem alatt az alkalmazás-ablakban egy állapotosor található. Az állapotosorban egy folyamatjelző és egy üzenetsáv van egymás mellett. A folyamatjelzőben jelezhetjük a hosszabb adatfeldolgozás állását, az üzenetsávban pedig szöveges üzeneteket jeleníthetünk meg a felhasználó tájékoztatására.

A GNOME programkönyvtár a `GnomeAppBar` típust biztosítja az alkalmazások alsó sorában megjelenő állapotosor kezelésére. Az állapotosorban egy folyamatjelző és egy szöveges üzenetsáv található egymás mellett, amelyeket egyszerű eszközökkel kezelhetünk.

39. példa. A következő példaprogram az állapotosorban található folyamatjelző használatát mutatja be.

```

1 void
2 on_progress_activate(GtkMenuItem *menuitem,
3                      gpointer user_data)
4 {
5     GtkWidget *appbar;
6     gfloat p;
7
8     appbar = lookup_widget(GTK_WIDGET(menuitem), "bar");
9     for (p = 0; p <= 1.0; p += 0.01){
10         gnome_appbar_set_progress_percentage(
11             GNOME_APPBAR(appbar), p);
12         gtk_main_iteration();
13         usleep (10000);
14     }
15 }
```

A program a 10–11. sorban beállítja a folyamatjelző által mutatott értéket, azonban ahhoz, hogy az új érték meg is jelenjen a képernyőn, a programnak vissza kell adnia a vezérlést, hogy a GTK+ programkönyvtár újrarajzolhassa a képernyőt. Ezt a `gtk_main_iteration()` függvény hívásával érjük el, amely feldolgozza a GTK+ számára összegyűlt üzeneteket, elvégzi a felgyülemlett feladatokat, majd visszaadja a vezérlést.

40. példa. Az üzenetsávban megjelenő szöveges üzeneteket veremszerűen tudjuk elhelyezni és eltávolítani. A következő két függvény bemu-

tatja hogyan tudunk egy üzenetet megjeleníteni és a legutóbbi üzenetet eltávolítani.

```

1  /*
2   * Az állapotsorban új üzenetet megjelenítő függvény.
3   */
4  void
5  on_appbar_message_activate(GtkMenuItem *menuitem,
6                             gpointer user_data)
7  {
8      GtkWidget *appbar;
9      static gint n = 1;
10     gchar *message;
11
12     appbar = lookup_widget(GTK_WIDGET(menuitem), "bar");
13     message = g_strdup_printf("Az %d. üzenet...", n);
14     gnome_appbar_push(GNOME_APPBAR(appbar), message);
15     g_free(message);
16     n++;
17 }
18
19 /*
20 * Az állapotsor legutóbbi üzenetét eltávolító függvény.
21 */
22 void
23 on_appbar_message_remove_activate(GtkMenuItem *menuitem,
24                                    gpointer user_data)
25 {
26     GtkWidget *appbar;
27
28     appbar = lookup_widget(GTK_WIDGET(menuitem), "bar");
29     gnome_appbar_pop(GNOME_APPBAR(appbar));
30 }

```

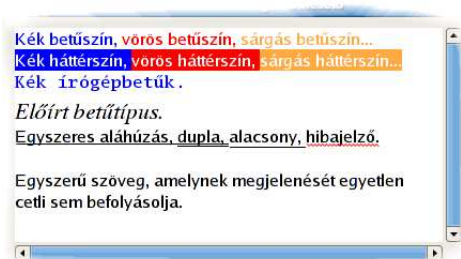
A példaprogram magától értetődő módon használja a GNOME két függvényét a szöveg megjelenítésére és eltávolítására a program 14., illetve 29. soraiban.

8.2. A szövegszerkesztő

A szövegszerkesztő többsoros szövegek, nagyobb állományok szerkesztésére használható képernyőelem, amely igen nagy méretű szöveg kezelésére is alkalmas, fejlett felhasználói felülettel rendelkező képernyőelem. A

GTK+ programkönyvtárban a szövegszerkesztő képernyőelem kezelésére a `GtkTextView` típus szolgál. A szövegszerkesztő képernyőelem meglehetősen összetett eszköz. A kezelésére használt függvényeket, típusokat, a tulajdonságait és szolgáltatásait egy-egy feladatcsoportra koncentrálni több részletben tárgyaljuk.

A szövegszerkesztőben megjelenő szöveg tárolására a szövegmemória, más néven szövegbuffer használható. Minden szövegszerkesztőhöz tartozik egy szövegmemória, de egy szövegmemóriához több szövegszerkesztő is tartozhat. Ez utóbbi esetben a szöveg a képernyő több helyén is feltűnhet, a felhasználó ugyanazt a szöveget több szövegszerkesztőben is szerkesztheti. A GTK+ programkönyvtárban a szövegmemória kezelésére a `GtkTextBuffer` típust használhatjuk.



8.4. ábra. A cetlik használata

A szövegmemória kezelésére sokszor használjuk a szövegbejárót, amely a szöveg egy adott pontjának kijelölésére használható a szövegmemóriában. A szövegbejáró segítségével „bejárhatjuk” a teljes szöveget, ami a szövegmemóriában található, a segítségével kijelölhetjük a szöveg egyes pontjait. A szövegbejárók fontos tulajdonsága, hogy a szövegmemória tartalmának megváltoztatásával érvényüket veszítik, azaz a szövegbejárókat „hosszú távú” munkára nem használhatjuk. A GTK+ programkönyvtárban a szö-

vegbejárók kezelésére a `GtkTextIter` típus szolgál.

Ha a szöveg egyes elemeit hosszú távon szeretnénk megjeleltetni, úgy, hogy a szöveg adott pontjára a szövegmemória változása után is rátaláljunk, a szövegjelet kell használnunk. A szövegbejáróval szemben a szövegjel a szövegmemória megváltozása után is használható marad. A GTK+ programkönyvtár a szövegjelek kezelésére a `GtkTextMark` típust biztosítja a programozó számára.

A szövegszerkesztő igen hatékony és magas szintű eszköz, ami többek közt abban is megnyilvánul, hogy képes különféle megjelenési formákat rendelni a szöveg egyes részeihez. A 8.4 ábrán láthatjuk, hogy a szövegszerkesztőben a szöveg egyes részei különféle előtér- és háttérzínekkel jelenhetnek meg, a szövegszerkesztő többféle betűtípust, betűváltozatot vagy éppen aláhúzást használhat egy időben. Az ilyen feladatokra a cetliket használhatjuk. A cetlik a szöveg egyes részeihez rendelhetők, hogy az adott szövegrész a cetlinek megfelelő tulajdonságokkal jelenjen meg. A GTK+ programkönyvtár a szövegben használható cetlik kezelésére a `GtkTextTag` struktúrát biztosítja a programozó számára.

8.2.1. A szöveg olvasása és írása

Ha szövegszerkesztő képernyőelemet használunk, akkor még a legegyszerűbb alkalmazásban is szükségünk van arra, hogy a szövegszerkesztőben megjelenő szöveghez hozzáférjünk, hogy a szövegszerkesztőbe szöveget helyezzünk el és a felhasználó által megváltoztatott tartalmat onnan kiolvassuk.

A szövegszerkesztő tartalmának beállítása viszonylag egyszerű, a szöveg kiolvasásához pedig a szövegszerkesztőhöz tartozó szövegmemóriában szövegbejárókat kell elhelyeznünk, hogy a köztük található szöveget kiolvassuk. A következő függvények szükségesek a szövegszerkesztő tartalmának beállítására és kiolvasására.

`GtkTextBuffer *gtk_text_view_get_buffer(GtkTextView *szövegszerkesztő);` A függvény segítségével lekérdezhetjük a szövegszerkesztőhöz tartozó szövegmemória címét. A legtöbb szövegkezelő műveletet a szövegmemórián tudjuk elvégezni.

A függvény argumentuma a szövegszerkesztőt jelöli a memóriában, a visszatérési értéke pedig a szövegmemória címét adja meg. Minden szövegszerkesztőhöz egy szövegmemória tartozik, de a szövegmemória egyszerre több szövegszerkesztőhöz lehet rendelve.

`void gtk_text_buffer_set_text(GtkTextBuffer *szmemória, const gchar *szöveg, gint hossz);` A függvény segítségével beállíthatjuk a szövegmemória tartalmát. A függvény törli a szövegmemóriából a benne található szöveget és bemásolja a második paraméter által jelölt szöveget. A harmadik paraméter a szöveg hosszát adja meg, ha az értéke `-1`, akkor a függvény feltételezi, hogy a szöveg végét `0` értékű bájt jelzi.

A szövegnek – mint a GTK+ könyvtár legtöbb függvénye esetében – UTF-8 kódolásúnak kell lennie.

`void gtk_text_buffer_get_start_iter(GtkTextBuffer *szmemória, GtkTextIter *bejáró);` A függvény segítségével egy létező szövegbejárót beállíthatunk úgy, hogy az a szövegmemóriában található szöveg legvégére – a lezáró `0` értékű bájtra – mutasson.

A függvény első argumentuma a szövegmemóriát jelöli a memóriában, a második pedig egy olyan memóriaterületet, ahol a szövegbejárónak foglaltunk helyet. A függvény a második argumentummal jelölt memóriaterületet módosítja, így annak írhatónak kell lennie.

`void gtk_text_buffer_get_end_iter(GtkTextBuffer *szmemória, GtkTextIter *bejáró);` A függvény segítségével

gével egy létező szövegbejárót állíthatunk be úgy, hogy az a szövegmemória legelső karakterét jelölje.

```
gchar *gtk_text_buffer_get_text(GtkTextBuffer *szmemória,
    const GtkTextIter *eleje, const GtkTextIter *vége,
    gboolean rejtett);
```

A függvény segítségével a szövegmemóriában a két bejáró közti szöveget másolhatjuk ki dinamikusan foglalt memóriaterületre.

A függvény első argumentuma a szövegmemóriát jelöli a memóriában, a második és harmadik argumentuma pedig a kimásolandó szöveg elejét, illetve végét jelölő szövegbejárót. A függvény utolsó argumentuma egy logikai érték, amely meghatározza, hogy a szöveg rejtett karakterei is jelenjenek -e meg a kimásolt területen.

A függvény visszatérési értéke a dinamikusan foglalt memóriaterületre mutat, ahova a függvény a kért szövegrészt kimásolta. Használat után ezt a memóriaterületet fel kell szabadítanunk.

Magától értetődik, hogy ha a függvénynek átadott két szövegbejáró a szövegmemória elejére és végére mutat, akkor a teljes szöveget kimásolhatjuk a szövegmemóriából.

```
void gtk_text_buffer_insert(GtkTextBuffer *szmemória,
    GtkTextIter *bejáró, const gchar *szöveg, gint hossz);
```

A függvény segítségével új szövegrészletet helyezhetünk el a szövegmemóriában található szövegben. A függvény paramétereinek értelmezése a következő:

szmemória Azt a szövegmemóriát jelöli a memóriában, amelyben az új szövegrészt be akarjuk szúrni.

bejáró Azt a szövegbejárót jelöli a memóriában, amely meghatározza, hogy szöveg mely részénél jelenjen meg az új szövegrész.

szöveg A beszúrandó karakterláncot jelöli a memóriában.

hossz A beszúrandó szövegrész hossza. Ha ennek a paraméternek -1 az értéke, a függvény feltételezi, hogy a beszúrandó szövegrész végét 0 értékű bájtt jelöli.

A függvény a beszúrandó szövegrészről másolatot készít, ezért a tárolására szolgáló memóriaterületet a későbbiekben felszabadíthatjuk.

A szövegszerkesztő képernyőelemben található szöveg beállítását és kiolvasását láthatjuk a következő két példában, amely ezeket a fontos lépéseket mutat be.

41. példa. A következő programrészlet a szövegszerkesztő mezőben található szöveget állományba írja.

```

1  int
2  save_textview_to_file(
3      GtkWidget *parent,
4      GtkTextView *text_view,
5      char *filename){
6
7      int output_file;
8      int written;
9      GtkWidget *dialog;
10     GtkTextBuffer *text_buffer;
11     GtkTextIter start;
12     GtkTextIter end;
13     gchar *text;
14     size_t size;
15
16     /*
17      * A kimenő állomány megnyitása.
18      */
19     output_file = open(filename,
20         O_WRONLY | O_CREAT,
21         S_IRUSR | S_IWUSR);
22
23     if (output_file == -1){
24         dialog = gtk_message_dialog_new(GTK_WINDOW(parent),
25             GTK_DIALOG_DESTROY_WITH_PARENT,
26             GTK_MESSAGE_ERROR,
27             GTK_BUTTONS_CLOSE,
28             "A fájl nem nyitható meg írásra: '%s': %m.",
29             filename);
30         gtk_dialog_run(GTK_DIALOG(dialog));
31         gtk_widget_destroy(dialog);
32         close(output_file);
33         return -1;
34     }
35
36     /*
37      * A szöveg kiolvasása a szövegszerkesztőből.
38      */
39     text_buffer = gtk_text_view_get_buffer(text_view);
40     gtk_text_buffer_get_start_iter(text_buffer, &start);
41     gtk_text_buffer_get_end_iter(text_buffer, &end);
42     text = gtk_text_buffer_get_text(text_buffer,
43         &start,

```

214

```

44         &end,
45         TRUE);
46     /*
47     * A szöveg írása. Az strlen() függvény itt nem a
48     * karakterek számát, hanem a foglalt memória méretét
49     * adja meg.
50     */
51     size = strlen(text);
52     if ((written=write(output_file, text, size)) != size){
53         dialog = gtk_message_dialog_new(GTK_WINDOW(parent),
54             GTK_DIALOG_DESTROY_WITH_PARENT,
55             GTK_MESSAGE_ERROR,
56             GTK_BUTTONS_CLOSE,
57             "Hiba az állomány írása közben: '%s': %m",
58             filename);
59         gtk_dialog_run(GTK_DIALOG(dialog));
60         gtk_widget_destroy(dialog);
61         written = -1;
62     }
63
64     /*
65     * A memóriaterület felszabadítása, az állomány
66     * lezárása.
67     */
68     g_free(text);
69     close(output_file);
70     return written;
71 }

```

A bemutatott függvény a megjegyzések segítségével könnyen megérthető, receptként felhasználható.

42. példa. A következő függvény beolvassa az állomány tartalmát és elhelyezi egy szövegszerkesztőben.

```

1  int
2  load_file_to_textview(GtkWidget *parent,
3                        GtkWidget *text_view,
4                        char *filename){
5      GtkWidget *dialog;
6      GtkTextBuffer *text_buffer;
7      struct stat for_size;
8      char *buffer;
9      int input_file;

```

```

10
11  /*
12   * A bemenő állomány méretének megállapítása.
13   */
14  if (stat(filename, &for_size) != 0){
15      dialog = gtk_message_dialog_new(GTK_WINDOW(parent),
16                                      GTK_DIALOG_DESTROY_WITH_PARENT,
17                                      GTK_MESSAGE_ERROR,
18                                      GTK_BUTTONS_CLOSE,
19                                      "A fájl mérete nem állapítható meg: '%s': %m",
20                                      filename);
21      gtk_dialog_run(GTK_DIALOG(dialog));
22      gtk_widget_destroy(dialog);
23      return -1;
24  }
25
26  /*
27   * Az állomány megnyitása.
28   */
29  buffer = g_malloc(for_size.st_size);
30  input_file = open(filename, O_RDONLY);
31  if (input_file == -1){
32      dialog = gtk_message_dialog_new(GTK_WINDOW(parent),
33                                      GTK_DIALOG_DESTROY_WITH_PARENT,
34                                      GTK_MESSAGE_ERROR,
35                                      GTK_BUTTONS_CLOSE,
36                                      "A fájl nem nyitható meg olvasásra: '%s': %m",
37                                      filename);
38      gtk_dialog_run(GTK_DIALOG(dialog));
39      gtk_widget_destroy(dialog);
40      g_free(buffer);
41      return -1;
42  }
43
44  /*
45   * Az állomány tartalmának beolvasása.
46   */
47  if (read(input_file, buffer, for_size.st_size) !=
48      for_size.st_size){
49      dialog = gtk_message_dialog_new(GTK_WINDOW(parent),
50                                      GTK_DIALOG_DESTROY_WITH_PARENT,
51                                      GTK_MESSAGE_ERROR,
52                                      GTK_BUTTONS_CLOSE,
53                                      "A fájl nem nyitható meg olvasásra: '%s': %m",

```

```

54     filename);
55     gtk_dialog_run(GTK_DIALOG(dialog));
56     gtk_widget_destroy(dialog);
57     close(input_file);
58     g_free(buffer);
59     return -1;
60 }
61
62 /*
63  * A szöveg elhelyezése a szövegszerkesztőben.
64  */
65 text_buffer = gtk_text_view_get_buffer(text_view);
66 gtk_text_buffer_set_text(text_buffer,
67     buffer,
68     for_size.st_size);
69
70 /*
71  * Az állomány rezárása, a memória felszabadítása.
72  */
73 close(input_file);
74 g_free(buffer);
75 return for_size.st_size;
76 }

```

A program – az előző példa programjához hasonló módon – könnyen megérthető, egyszerű lépéseket tartalmaz.

8.2.2. A szövegbejárók

A következőkben ismertetjük a szövegbejárók – a `GtkTextIter` struktúrák – kezelésére használható függvényeket. A szövegbejárókat kezelő függvények kapcsán meg kell jegyeznünk, hogy a „szövegsor” kifejezés némiképpen különböző jelentésű lehet a szövegmemória és a szövegszerkesztő képernyőelem szempontjából. A szövegszerkesztő képernyőelem ugyanis a korlátozott szélessége miatt sokszor több sorra tördeli a szöveget, ami a szövegmemóriában egyetlen sorban található. A szövegmemória sorait a köznap értelemben tehát inkább bekezdéseknek tekinthetjük, olyan szövegegységeknek, amelyeknek az elején és a végén leütöttük az Enter billentyűt (azaz ahová *újsor* karaktert helyeztünk, hogy még zarosabb legyen a sorok és bekezdések közti különbség).

Érdemes azt is megjegyeznünk, hogy a szövegbejárók mozgatására használható függvények mindig hamis logikai értéket adnak vissza, ha a szövegbejáró a mozgatás során elérte a szövegmemória végét, de nem

feltétlenül adnak vissza hamis értéket, ha a szövegbejáró elérte a szövegmemória elejét. Ezt a látszólag értelmetlen asszimetriát az magyarázza, hogy a szövegbejáró általában az utána következő karaktert jellemzi, és a szövegmemória végén található szövegbejáró után nyilvánvalóan nem található egyetlen karakter sem, ami nem mindig mondhatunk el a szövegmemória elejére mozgatott szövegbejáró kapcsán.

A GTK+ programkönyvtár a következő függvényeket biztosítja a szövegbejárók használatára:

`GtkTextBuffer *gtk_text_iter_get_buffer(const GtkTextIter *bejáró);` A függvény segítségével lekérdezhetjük, hogy a szövegbejáróhoz melyik szövegmemória tartozik.

`gint gtk_text_iter_get_offset(const GtkTextIter *bejáró);` A függvény segítségével lekérdezhetjük, hogy az adott szövegbejáró hányadik karakternél található. A karakterek számozása 0-tól kezdődik.

`gint gtk_text_iter_get_line(const GtkTextIter *bejáró);` A függvény segítségével lekérdezhetjük, hogy a szövegbejáró hányadik sorban található. A sorok számozása 0-tól kezdődik.

`gint gtk_text_iter_get_line_offset(const GtkTextIter *bejáró);` A függvény segítségével lekérdezhetjük, hogy a szövegbejáró a soron belül hányadik karakternél található. A karakterek számozása a soron belül is 0-tól kezdődik.

`gint gtk_text_iter_get_line_index(const GtkTextIter *bejáró);` A függvény segítségével lekérdezhetjük, hogy a szövegbejáró a soron belül hányadik bájt nál található. (A szövegmemória UTF-8 kódolást használ, ahol nem minden karakter kódja 1 bájt os.) Az adatbájtok számozása 0-tól kezdődik.

`gunichar gtk_text_iter_get_char(const GtkTextIter *bejáró);` A függvény segítségével lekérdezhetjük a szövegbejárónál található karaktert. A függvény a szövegbejárót nem mozgatja.

`gchar *gtk_text_iter_get_text(const GtkTextIter *eleje, const GtkTextIter *vége);` A függvény segítségével lekérhetjük a szövegmemóriából a szövegbejárók közti szöveget. A függvény visszatérési értéke egy olyan mutató, amely arra a dinamikusan foglalt memóriaterületre mutat, amelyre a függvény kimásolta a szövegbejárók közti szövegrészletet.

`GSList *gtk_text_iter_get_marks(const GtkTextIter *bejáró);` A függvény segítségével lekérhetjük, a szövegmemóriában az adott szövegbejáróval egyező helyen található szövegelemek

listáját. A függvény visszatérési értéke egyszeresen láncolt listát jelöl a memóriában, amelyben adatként megtalálhatók mindazok a szövegjelek, amelyek a szövegmemória adott helyén találhatók.

GSList *gtk_text_iter_get_toggled_tags(const GtkTextIter *bejáró, gboolean bekapcsolva); A függvény segítségével lekérdezhetjük a szövegmemória adott területére vonatkozó cetlik listáját. Ha a függvény második paramétere **TRUE**, akkor a függvény azoknak a cetliknek a listáját adja, amelyek az adott ponton be vannak kapcsolva, ha **FALSE**, akkor azokat, amelyek az adott helyen nincsenek bekapcsolva.

A függvény visszatérési értéke egyszeresen láncolt listát jelöl a memóriában, amely adatként tartalmazza a cetliket, amelyek az adott ponton kifejtik hatásukat.

gboolean gtk_text_iter_begins_tag(const GtkTextIter *bejáró, GtkTextTag *cetli); A függvény segítségével lekérdezhetjük, hogy a szövegmemória adott pontján kezdődik-e a megadott cetli hatása a szövegre. Ha a függvény visszatérési értéke **TRUE** az azt jelenti, hogy a szövegbejáró előtt a megadott cetli nincs hatással a szövegre, utána azonban igen.

Ha a függvénynek második paraméterként nem egy cetli címét, hanem **NULL** értéket adunk meg, akkor a függvény igaz értéket ad vissza, ha *bármelyik* cetli hatása az adott ponton kezdődik.

gboolean gtk_text_iter_ends_tag(const GtkTextIter *bejáró, GtkTextTag *cetli); A függvény használata megegyezik a **gtk_text_iter_begins_tag()** függvény hatásával, de akkor ad igaz értéket, ha a cetli hatása az adott ponton fejeződik be.

gboolean gtk_text_iter_toggles_tag(const GtkTextIter *bejáró, GtkTextTag *cetli); A függvény igaz értéket ad vissza, ha a szövegmemória adott pontján az adott cetli hatása kezdődik vagy befejeződik. Ha a függvény második paramétere nem egy cetlit jelöl a memóriában, hanem **NULL** értéket képvisel, a függvény igaz értéket ad, ha *bármelyik* cetli hatása az adott ponton kezdődik vagy fejeződik be.

gboolean gtk_text_iter_has_tag(const GtkTextIter *bejáró, GtkTextTag *cetli); A függvény segítségével megvizsgálhatjuk, hogy a szövegmemória adott helyén a megadott cetli kifejti-e a hatását. Ha a cetli a szöveg adott pontján kifejti a hatását, a függvény **TRUE** értéket ad vissza, különben pedig **FALSE** a visszatérési érték.

GSList *gtk_text_iter_get_tags(const GtkTextIter *bejáró); A függvény segítségével lekérdezhetjük a szöveg adott pontjára érvényes cetlik listáját. A függvény visszatérési értéke egy olyan egyszeresen láncolt listát jelöl a memóriában, amely adatként a szöveg adott pontjára érvényes cetliket tartalmazza. A listában a cetlik növekvő fontossági sorrendben találhatók meg.

gboolean gtk_text_iter_starts_word(const GtkTextIter *bejáró); A függvény igaz értéket ad vissza, ha a szövegbejáró szó elején található.

gboolean gtk_text_iter_ends_word(const GtkTextIter *bejáró); A függvény igaz értéket ad vissza, ha a szövegbejáró szó végén található.

gboolean gtk_text_iter_inside_word(const GtkTextIter *bejáró); A függvény igaz értéket ad vissza, ha a szövegbejáró szó belsejében található.

gboolean gtk_text_iter_starts_line(const GtkTextIter *bejáró); A függvény igaz értéket ad vissza, ha a szövegbejáró sor elején található.

gboolean gtk_text_iter_ends_line(const GtkTextIter *bejáró); A függvény igaz értéket ad vissza, ha a szövegbejáró a sor végén van.

gboolean gtk_text_iter_starts_sentence(const GtkTextIter *bejáró); A függvény igaz értéket ad vissza, ha a szövegbejáró mondat elején található.

gboolean gtk_text_iter_ends_sentence(const GtkTextIter *bejáró); A függvény igaz értéket ad vissza, ha a szövegbejáró mondat végén található.

gboolean gtk_text_iter_inside_sentence(const GtkTextIter *bejáró); A függvény igaz értéket ad vissza, ha a szövegbejáró mondaton belül található.

gboolean gtk_text_iter_is_cursor_position(const GtkTextIter *bejáró); A függvény igaz értéket ad vissza, ha a szövegbejáró olyan helyen van, ahol a kurzor is megjelenhet. A függvény tehát nem azt jelzi, hogy a szövegbejáró és a kurzor ugyanazon a helyen van, hanem azt, hogy a kurzor akár ott is lehetne, ahol a szövegbejáró éppen van.

gint gtk_text_iter_get_chars_in_line(const GtkTextIter *bejáró); A függvény segítségével lekérdezhetjük a karakterek számát abban a sorban, amelyben a szövegbejáró is van. A karakterek számába az újsor karakter is beleszámít.

gint gtk_text_iter_get_bytes_in_line(const GtkTextIter *bejáró); A függvény segítségével lekérdezhetjük, hogy hány bájtnyi szöveg van a sorban, amelyben a szövegbejáró is található. (Az UTF-8 kódolást használva a karakterek száma és az adatbájtok száma nem feltétlenül egyezik meg egymással.) Az adatbájtok számába az újsorkarakter által foglalt adatbájtok száma is beletartozik.

gboolean gtk_text_iter_is_end(const GtkTextIter *bejáró); A függvény igaz értéket ad vissza, ha a szövegbejáró a szövegmemória legvégén található.

gboolean gtk_text_iter_is_start(const GtkTextIter *bejáró); A függvény igaz értéket ad vissza, ha a szövegbejáró a szövegmemória elején található.

gboolean gtk_text_iter_forward_char(GtkTextIter *bejáró); A függvény egy karakterrel mozgatja a szövegbejárót a szöveg vége felé. A függvény visszatérési értéke igaz, ha a mozgatus sikeres volt – azaz a szövegbejáró nem a szövegmemória végén volt.

gboolean gtk_text_iter_backward_char(GtkTextIter *bejáró); A függvény egy karakterrel mozgatja a szövegbejárót a szöveg eleje felé. A függvény visszatérési értéke igaz, ha a mozgatus sikeres volt, hamis, ha a szövegbejáró már a szövegmemória első karakterénél előtt található.

gboolean gtk_text_iter_forward_chars(GtkTextIter *bejáró, gint n); A függvény segítségével a szövegbejárót tetszőleges számú karakterrel mozgathatjuk előre, a szöveg vége felé. A függvény **TRUE** értéket ad vissza, ha a szövegbejárót a kívánt számú karakterrel sikerült a szöveg vége felé mozgatni, **FALSE** értéket, ha a mozgatus során a szövegbejáró elérte a szövegmemória végét.

gboolean gtk_text_iter_backward_chars(GtkTextIter *bejáró, gint n); A függvény hasonlóan működik a **gtk_text_iter_forward_chars()** függvényhez, de nem a szöveg vége, hanem a szöveg eleje felé mozgatja a szövegbejárót.

gboolean gtk_text_iter_forward_line(GtkTextIter *bejáró); A függvény segítségével a szövegbejárót a szövegmemória következő sorának elejére mozgathatjuk. A függvény **TRUE** értéket ad vissza,

ha a szövegmemóriában a következő sor elejére lehetett mozgatni a szövegbejárót, `FALSE` értéket, ha a szövegmemóriában már nincsen következő sor. Ez utóbbi esetben a függvény a szövegbejárót a szövegmemória végére mozgatja.

`gboolean gtk_text_iter_backward_line(GtkTextIter *bejáró);` A függvény segítségével a szövegbejárót a szövegmemória előző sorának elejére mozgathatjuk. Ha a szövegbejáró a szövegmemória első sorában található, de nem a sor elején, a függvény a szövegbejárót a sor elejére mozgatja. A függvény `TRUE` értéket ad vissza, ha a szövegbejáró a függvény hatására elmozdult, `FALSE` értéket, ha a szövegbejáró a függvény hívásakor már az első sor elején volt.

Vegyük észre, hogy a függvény akkor is adhat vissza igaz értéket, ha a szövegbejárót nem sikerült az előző sorba mozgatni, csak az adott sor elejére. Ez némiképpen megnehezítheti a programozó munkáját, akinek erre az eshetőségre is gondolnia kell.

`gboolean gtk_text_iter_forward_lines(GtkTextIter *bejáró, gint n);` A függvény használata és működése megegyezik a `gtk_text_iter_forward_line()` függvény használatával és működésével, de nem egy, hanem tetszőleges sornyi mozgatót tesz lehetővé.

`gboolean gtk_text_iter_backward_lines(GtkTextIter *bejáró, gint n);` A függvény használata és működése megegyezik a `gtk_text_iter_backward_line()` függvény használatával és működésével, de nem egy, hanem tetszőleges sornyi mozgatót tesz lehetővé.

`gboolean gtk_text_iter_forward_word_end(GtkTextIter *bejáró);` A függvény segítségével a szövegbejárót a szó végére mozgathatjuk. Ha a szövegbejáró már a szó végén található, akkor a függvény a szövegbejárót a következő szó végére mozgatja, így a függvényt újra és újra hívva a teljes szövegmemóriát bejárhatjuk.

A függvény `TRUE` értéket ad vissza, ha a szövegbejárót sikerült elmozdítani és az a mozgató után még nem érte el a szövegmemória végét.

`gboolean gtk_text_iter_backward_word_start(GtkTextIter *bejáró);` A függvény segítségével a szövegbejárót a szó elejére mozgathatjuk. Ha a szövegbejáró a függvény hívásakor már a szó elején volt, a függvény a szövegbejárót az előző szó elejére mozgatja, így a függvényt újra és újra hívva a teljes szöveg bejárható.

222

A függvény **TRUE** értéket ad vissza, ha a szövegbejárót sikerült elmozdítani.

gboolean gtk_text_iter_forward_word_ends(GtkTextIter *bejáró, gint n); A függvény használata és működése a **gtk_text_iter_forward_word_end()** függvény használatához és működéséhez hasonló, de nem egy, hanem tetszőleges számú szónyi mozgatót tesz lehetővé.

gboolean gtk_text_iter_backward_word_starts(GtkTextIter *bejáró, gint count); A függvény használata és működése a **gtk_text_iter_forward_word_start()** függvény használatához és működéséhez hasonló, de nem egy, hanem tetszőleges számú szónyi mozgatót tesz lehetővé.

gboolean gtk_text_iter_forward_cursor_position(GtkTextIter *bejáró); A függvény segítségével a szövegbejárót egy kurzorpozícióval mozgathatjuk a szöveg vége felé. A függvény működése nem teljesen egyezik meg a **gtk_text_iter_forward_char()** függvény működésével, mert a karakter és a kurzorpozíció fogalma bizonyos esetekben nem azonos. Bizonyos nyelvek néhány írásjele több karakterből áll, ahol kurzor nem állhat az írásjelet alkotó karakterek közé.

A függvény **TRUE** értéket ad vissza, ha a szövegbejáró mozgatása sikeres volt és a mozgató során nem értük el a szövegmemória végét.

gboolean gtk_text_iter_backward_cursor_position(GtkTextIter *bejáró); A függvény segítségével a szövegbejárót egy kurzorpozícióval mozgathatjuk a szöveg eleje felé. (A kurzorpozíció fogalmát lásd az előző függvénynél.)

A függvény **TRUE** értéket ad vissza, ha a szövegbejáró mozgatása sikeres volt.

gboolean gtk_text_iter_forward_cursor_positions(GtkTextIter *iter, gint count); A függvény működése és használata megegyezik a **gtk_text_iter_forward_cursor_position()** függvény működésével és használatával, de nem egy, hanem tetszőlegesen sok kurzorpozícióval mozgathatjuk a szövegbejárót.

gboolean gtk_text_iter_backward_cursor_positions(GtkTextIter *bejáró, gint count); A függvény működése és használata megegyezik a **gtk_text_iter_backward_cursor_position()** függvény működésével és használatával, de nem egy, hanem tetszőlegesen sok kurzorpozícióval mozgathatjuk a szövegbejárót.

gboolean gtk_text_iter_backward_sentence_start(GtkTextIter *bejáró); A függvény segítségével a bejáró a mondat elejére mozgatható. Ha a bejáró már a mondat elején van, a függvény az előző mondat elejére kerül. A függvény visszatérési értéke **TRUE**, ha a bejárót sikerült elmozdítani.

gboolean gtk_text_iter_forward_sentence_end(GtkTextIter *bejáró); A függvény segítségével a bejáró a mondat végére mozgatható. Ha a bejáró már a függvény hívásakor is a mondat végén található, a függvény a bejárót a következő mondat végére mozgatja. A függvény **TRUE** értéket ad vissza, ha a bejárót sikerült elmozdítani és az a mozgatás során nem érte el a szövegmémória végét.

gboolean gtk_text_iter_backward_sentence_starts(GtkTextIter *bejáró, gint count); A függvény használata megegyezik a **gtk_text_iter_backward_sentence_start()** függvény használatával, de lehetővé teszi, hogy a bejárót több mondatnyi távolságba mozgassuk.

gboolean gtk_text_iter_forward_sentence_ends(GtkTextIter *bejáró, gint count); A függvény használata megegyezik a **gtk_text_iter_forward_sentence_start()** függvény használatával, de lehetővé teszi, hogy a bejárót több mondatnyi távolságba mozgassuk.

void gtk_text_iter_set_offset(GtkTextIter *bejáró, gint számláló); A függvény segítségével a bejáró a szövegmémória adott sorszámú karakteréhez mozgatható. A függvény második paramétere meghatározza, hogy a szövegmémória hányadik karakteréhez mozgatjuk a szövegbejárót. A karakterek számlálása 0-tól indul.

void gtk_text_iter_set_line(GtkTextIter *bejáró, gint sor_száma); A függvény segítségével a szövegbejárót a szövegmémória adott sorszámú sorának elejére mozgathatjuk. A sorok számozása 0-tól indul.

void gtk_text_iter_set_line_offset(GtkTextIter *bejáró, gint karakter_szám); A függvény segítségével a szövegbejárót az aktuális soron belül, annak adott sorszámú karakterére mozgathatjuk. A soron belüli karakterek számozása 0-tól indul.

void gtk_text_iter_set_line_index(GtkTextIter *bejáró, gint bájt_szám); A függvény segítségével a szövegbejárót az aktuális soron belül, annak adott sorszámú bájtjára mozgathatjuk. A soron belüli bájtok számozása 0-tól indul.

224

void gtk_text_iter_forward_to_end(GtkTextIter *bejáró); A függvény segítségével a bejárót a szövegmémória végére mozgathatjuk.

gboolean gtk_text_iter_forward_to_line_end(GtkTextIter *bejáró); A függvény segítségével a bejárót a sor végére mozgathatjuk. Ha a szövegbejáró a függvény hívásakor már a sor végén található, a függvény a bejárót a következő sor végére mozgatja. A függvény **TRUE** értéket ad vissza, ha a bejáró mozgatása sikeres volt és a bejáró a mozgatás során nem érte el a szövegmémória végét.

gboolean gtk_text_iter_forward_to_tag_toggle(GtkTextIter *bejáró, GtkTextTag *tag); A függvény segítségével a bejárót a szöveg vége felé, arra a helyre mozgathatjuk, ahol az adott cetli hatóköre kezdődik vagy befejeződik. Ha a szövegbejáró aktuális helyén kezdődik vagy fejeződik be az adott cetli hatóköre, a függvény a következő hasonló pontra mozgatja a bejárót. Ha a függvény második paraméterének értéke **NULL** érték, a függvény nem egy konkrét cetli hatókörének elejét vagy végét keresi, egyszerűen csak a legközelebbi, tetszőleges cetli hatókörének elejét vagy végét keresi.

A függvény **TRUE** értéket ad vissza, ha a keresés sikeres volt és a bejárót sikerült elmozdítani.

gboolean gtk_text_iter_backward_to_tag_toggle(GtkTextIter *bejáró, GtkTextTag *cetli); A függvény működése és használata megegyezik a **gtk_text_iter_forward_to_tag_toggle()** működésével és használatával, de ez a függvény a szöveg eleje felé keres.

gboolean gtk_text_iter_forward_find_char(GtkTextIter *bejáró, GtkTextCharPredicate *függvény, gpointer adat, const GtkTextIter *határ); A függvény segítségével a szövegmémóriában adott karakter vagy karakterosztály következő előfordulását kereshetjük meg a szövegbejáró aktuális helyétől a szöveg vége felé haladva. A függvény paramétereinek értelmezése a következő:

bejáró A függvény ezt a bejárót mozgatja a szöveg vége felé, ennek a bejárónak az aktuális helyétől keres előre felé.

függvény A keresés során ez a függvény fogja eldönteni, hogy a keresett karaktert megtaláltuk-e. Ennek a függvénynek **TRUE** értéket kell visszaadni, ha a keresett karaktert megtaláltuk.

adat A függvény, amely a keresett karaktert azonosítja, megkapja ezt a mutatót is paraméterként.

határ A függvény addig végzi a keresést, amíg ezt a keresési határt el nem éri. Ha a függvény ezen paraméterének értéke **NULL**, a függvény a keresést a szövegmemória végéig folytatja.

viSSzatérési_érték A függvény viSSzatérési értéke **TRUE**, ha a keresés sikerrel járt, azaz, ha a függvény azért fejezte be, mert a karakter azonosítását végző függvény **TRUE** értéket adott vissza.

gboolean GtkTextCharPredicate(gunichar karakter, gpointer adat); A karakter vagy karakterosztály azonosítására szolgáló függvény, amelyet a programozó biztosít a kereséshez. A függvény első paramétere a vizsgálandó karakter, a második paramétere pedig a keresés elején megadott mutató, amely tetszőleges adatterületet jelöl a memóriában.

A függvénynek **TRUE** értéket kell visszaadnia, ha a karaktert azonosította.

gboolean gtk_text_iter_backward_find_char(GtkTextIter *bejáró, GtkTextCharPredicate *függvény, gpointer adat, const GtkTextIter *határ); A függvény működése és használata megegyezik a **gtk_text_iter_forward_find_char()** függvény működésével és használatával, de ez a függvény a szöveg eleje felé keres.

gboolean gtk_text_iter_forward_search(const GtkTextIter *bejáró, const gchar *szöveg, GtkTextSearchFlags kapcsolók, GtkTextIter *kezdet, GtkTextIter *vége, const GtkTextIter *határ); A függvény segítségével a szövegmemóriában a szöveg vége felé kereshetünk adott szövegrészletet. A függvény paramétereinek értelmezése a következő:

bejáró A keresés kezdőpontját jelölő szövegbejáró.

szöveg A keresett karakterlánc.

kapcsolók A kapcsolók, amelyek meghatározzák a keresés módját. Mivel a GTK+ jelenlegi változatai nem támogatják a láthatatlan szövegrészek használatát és e könyvben nem foglalkozunk a szövegbe ágyazott képernyőelemekkel, ennek a paraméternek a helyén egyszerűen 0 értéket adunk meg.

kezdet A szövegbejáró helye a memóriában, ahová a függvény a megtalált szövegrész elejének helyét elhelyezi.

vége A szövegbejáró helye a memóriában, ahová a függvény a megtalált szövegrész végének helyét elhelyezi.

határ A keresés felső határát jelző szövegbejáró helye a memóriában. Ha ennek a paraméternek **NULL** az értéke, a keresés a szövegmemória végéig tart.

visszatérési_érték A függvény **TRUE** értéket ad vissza, ha a keresés sikeres volt, azaz a függvény talált egyező szövegrészt.

gboolean gtk_text_iter_backward_search(const GtkTextIter *bejáró, const gchar *szöveg, GtkTextSearchFlags kapcsolók, GtkTextIter *kezdet, GtkTextIter *vége, const GtkTextIter *határ); A függvény működése és használata megegyezik a **gtk_text_iter_forward_search()** függvény működésével és használatával, de ez a függvény a szöveg eleje felé keres.

gboolean gtk_text_iter_equal(const GtkTextIter *a, const GtkTextIter *b); A függvény igaz értéket ad vissza, ha a paraméterként megadott szövegbejárók a szövegmemória egyazon helyére mutatnak. A GTK+ dokumentációja szerint ez a függvény igen hatékony, gyors algoritmust használ.

gint gtk_text_iter_compare(const GtkTextIter *a, const GtkTextIter *b); A szövegbejárókat összehasonlító függvény. A függvény 0 értéket ad vissza, ha a két szövegbejáró egyazon helyet jelöl a szövegmemóriában, negatív értéket ad vissza, ha az első szövegbejáró a szöveg elejéhez közelebb, pozitív értéket, ha távolabb van.

gboolean gtk_text_iter_in_range(const GtkTextIter *bejáró, const GtkTextIter *eleje, const GtkTextIter *vége); A függvény **TRUE** értéket ad vissza, ha az első szövegbejáró a második és harmadik paraméterként megadott szövegbejárók közt található. A második és harmadik szövegbejárónak a megfelelő sorrendben kell lennie, azaz a harmadik paraméterként megadott szövegbejárónak a szöveg végéhez közelebb kell lennie, mint a második paraméterként megadott szövegbejárónak.

void gtk_text_iter_order(GtkTextIter *a, GtkTextIter *b); A függvény a megadott két szövegbejárót sorrendbe állítja, azaz, ha szükséges megcseréli, hogy a második szövegbejáró közelebb legyen a szövegmemória végéhez, mint az első.

8.2.3. A szövegjelek

Amint azt már említettük, a szövegjelek – a **GtkTextMark** struktúrák – a szöveg egyes helyeinek hosszú távú megjelölésére használhatók. A **GtkTextMark** a **GtkTextIter** típushoz hasonlóan használható, azzal el-lététben azonban a szöveg megváltoztatása után is érvényes marad. Amíg

a `GtkTextIter` adatszerkezetet általában egy függvényen belül használjuk – ott hozzuk létre és ott is semmisítjük meg, amíg a szöveg nem változhat meg –, addig a `GtkTextMark` élettartama általában több függvényt is átöl – nem abban a függvényben semmisítjük meg, amelyikben létrehozuk. A `GtkTextMark` ezen tulajdonsága több, a kezelésére szolgáló függvény kapcsán megfigyelhető, a `GtkTextMark` adatszerkezetet névvel léthatjuk el, dinamikusan foglalt memóriaterületre kerül, tulajdonságai közt a szöveg megváltozása esetén követendő eljárás is szerepel és így tovább.

A `GtkTextMark` adatszerkezet használatához elengedhetetlenül szükséges függvények a következők:

```
GtkTextMark *gtk_text_buffer_create_mark(GtkTextBuffer
*szmemória, const gchar *név, const GtkTextIter
*bejáró, gboolean balra);
```

A függvény segítségével új szövegjelet hozhatunk létre a szövegmemória adott pontjának jelölésére. A függvény visszatérési értékének és paramétereinek jelentése a következő:

visszatérési érték A függvény visszatérési értéke az újonan létrehozott szövegjelet jelöli a memóriában.

szmemória A szövegmemóriát jelöli a memóriában. A szövegjelet ennek a szövegmemóriának egy pontját jelöli, ehhez a szövegmemóriához tartozik. A szövegmemóriában a GTK nyilvántartást vezet az összes szövegjelről, a szövegmemóriában található összes szövegjel kikereshető, lekérdezhető.

név A szövegjel nevét jelöli a memóriában. A név alapján a szövegjel a szövegmemóriából kikereshető, mert a szövegmemóriában adott egy időben mindig csak egy szövegjel létezhet.

Ha ennek a paraméternek az értéke `NULL`, a szövegjelet név nélkül hozzuk létre.

bejáró A szövegjel helyét jelöli a memóriában. A szövegjelet tehát egy szövegbejáró segítségével hozhatjuk létre e függvénnyel.

balra Ennek a paraméternek a segítségével megadhatjuk, hogy mi történjék, ha pontosan ennek a szövegjelnek a helyére illesztünk be új karaktereket.

Ha ez a paraméter `TRUE` értékű, a beszűrt karakterektől balra lesz megtalálható a szövegjel. Ha e paraméternek az értéke `FALSE` a szövegjel a beszűrt karaktertől jobbra található, az új karakterek a szövegjel bal oldalára kerülnek, a szövegjelet „maguk előtt tolva”. Ilyen módon viselkedik például az a szövegjel is, ami a kurzor helyét jelöli a memóriában, hiszen a kurzor

mindig az új karaktertől jobbra tolódik – legalábbis a balról jobbra olvasandó írások esetén¹.

void gtk_text_buffer_delete_mark(GtkTextBuffer *szmemória, GtkTextMark *szövegjel); A függvény segítségével a szövegjel törölhető a szövegmemóriából. A szövegjel törlése után a szövegjel „törölt” állapotba kerül, tovább már nem használható.

A szövegjel törlése során a függvény a `g_object_unref()` függvény segítségével csökkenti a szövegjel hivatkozásszámlálóját. Ha ezt a számlálót nem növeltük a `g_object_ref()` függvénnyel, a `gtk_text_buffer_delete_mark()` hatására a hivatkozásszámláló 0-ra csökken, a szövegjel megsemmisül, a tárolására szolgáló memóriaterület felszabadul.

A szövegmemóriából törölt szövegjelek visszaállítására nincs mód.

A függvény első paramétere a szövegmemóriát jelöli a memóriában, amelyekhez a szövegjel tartozik, második paramétere pedig magát a szövegjelet.

void gtk_text_buffer_delete_mark_by_name(GtkTextBuffer *szmemória, const gchar *név); A függvény segítségével a szövegjelet nevének megadásával törölhetjük, azaz hatása ugyanaz, mint a `gtk_text_buffer_delete_mark()` függvényé, a szövegjelre azonban nevével nem pedig címével hivatkozhatunk.

A függvény első paramétere a szövegmemóriát jelöli a memóriában, amelyben a szövegjel található, második paramétere pedig a szövegjel nevének helyét adja meg.

void gtk_text_buffer_get_iter_at_mark(GtkTextBuffer *szmemória, GtkTextIter *bejáró, GtkTextMark *szövegjel); A függvény a szövegjelek és szövegbejárók közti átalakítást valósítja meg, segítségével szövegbejárót készíthetünk ami a szövegnek pontosan arra a pontjára mutat, ahol a szövegbejáró található.

A függvény első paramétere a szövegmemóriát, második paramétere pedig a beállítandó szövegbejárót jelöli a memóriában. A második paraméternek a memóriának olyan pontjára kell mutatnia, ahová a függvény egy szövegjel adatszerkezetet elemeit írhatja.

A függvény harmadik paramétere a szövegjelet jelöli a memóriában.

¹A szövegszerkesztő képernyőelem balról jobbra olvasandó nyelvek kezelésére is képes, sőt a szöveg iránya szakaszlól-szakaszra változhat is.

`GtkTextMark *gtk_text_buffer_get_mark(GtkTextBuffer *szmemória, const gchar *név);` A függvény segítségével a szövegmemóriához tartozó szövegjelet kereshetjük ki a névnek alapján a memóriából.

A függvény első paramétere a szövegmemóriát, második paramétere a szövegjel nevét jelöli a memóriában. A függvény visszatérési értéke a szövegjelet jelöli a memóriában, ha a keresés sikeres volt, `NULL` érték, ha nem.

`void gtk_text_buffer_move_mark(GtkTextBuffer *szmemória, GtkTextMark *szövegjel, const GtkTextIter *bejáró);` A függvény segítségével meglévő szövegjelet mozgathatunk a szöveg új pontjára.

A függvény első paramétere a szövegmemóriát, második paramétere az elmozdítandó szövegjelet jelöli a memóriában.

A függvény harmadik paraméterének egy szövegbejárót kell jelölnie a memóriában, ami a szövegnek arra a helyére mutat, ahová a szövegjelet át akarjuk helyezni.

`void gtk_text_buffer_move_mark_by_name(GtkTextBuffer *szmemória, const gchar *név, const GtkTextIter *bejáró);` A függvény a `gtk_text_buffer_move_mark()` függvényhez hasonlóan a szövegjel mozgatására használható, azzal szemben azonban a szövegjelre nevével, nem pedig címével hivatkozhatunk.

`GtkTextBuffer *gtk_text_mark_get_buffer(GtkTextMark *szövegjel);` A függvény segítségével lekérdezhetjük, hogy a szövegjel melyik szövegmemóriához tartozik.

Megfigyelhetjük, hogy a szövegjelek kezelésére használatos függvények mindegyikének paraméterként meg kell adnunk a szövegmemória címét is, ez azonban a szövegjel címének ismeretében ezzel a függvénnyel lekérdezhető. A szövegjelek kezelése közben tehát elegendő lehet magának a szövegjel címének nyilvántartása.

A függvény paramétere a szövegjelet jelöli a memóriában, visszatérési értéke pedig azt a szövegmemóriát, amelyben a szövegjel megtalálható.

A szövegjelek legfontosabb tulajdonságainak lekérdezésére és megváltoztatására a következő függvényeket használhatjuk.

`gboolean gtk_text_mark_get_deleted(GtkTextMark *szövegjel);` A függvény segítségével lekérdezhetjük, hogy a szövegjel törölt állapotban van-e. Ne felejtsük azonban

el, hogy ha a szövegjel hivatkozásszámlálóját nem növeltük, akkor a `gtk_text_buffer_delete_mark()` vagy a `gtk_text_buffer_delete_mark_by_name()` függvény hívásának hatására a szövegjel nem csak egyszerűen törölt állapotba kerül, hanem meg is semmisül.

A függvény paramétere a szövegjelet jelöli a memóriában, visszatérési értéke pedig megadja, hogy a szövegjel törölve van-e.

`const gchar *gtk_text_mark_get_name(GtkTextMark *szövegjel);` A függvény segítségével lekérdezhető a szövegjel neve.

A függvény paramétere a szövegjelet jelöli a memóriában, a visszatérési értéke pedig a szövegjelhez tartozó nevet jelöli a memóriában vagy `NULL`, ha a szövegjelnek nincs neve.

A visszatérési érték által jelzett memóriaterületet nem szabad módosítani.

`void gtk_text_mark_set_visible(GtkTextMark *szövegjel, gboolean látható);` A függvény segítségével beállíthatjuk, hogy a szövegjel látható legyen-e, megjelenjen-e a képernyőn. A látható szövegjelek helyén egy függőleges vonal jelenik meg a képernyőn. Az ilyen vonalak különösen hasznosak a program belövése során.

A függvény első paramétere a szövegjelet jelöli a memóriában, második paraméterének értéke pedig `TRUE` vagy `FALSE`, aszerint, hogy a szövegjelet meg akarjuk-e jeleníteni, illetve el akarjuk-e rejteni.

`gboolean gtk_text_mark_get_visible(GtkTextMark *szövegjel)` A függvény segítségével lekérdezhetjük, hogy a szövegjel látható-e.

A függvény paramétere a szövegjelet jelöli a memóriában, visszatérési értéke pedig megadja, hogy a szövegjel helyén a GTK+ programkönyvtárnak egy függőleges vonalat kell-e megjelenítenie – feltéve persze, hogy a szövegjelhez tartozó szöveg éppen látszik a képernyőn.

`gboolean gtk_text_view_move_mark_onscreen(GtkTextView *szövegszerkesztő, GtkTextMark *szövegjel);` A függvény segítségével a szövegjelet a képernyőn látható szövegterületre mozgathatjuk.

A szövegszerkesztő képernyőelemet általában görgetősávokkal mozgathatóan helyezzük el a képernyőn, így előfordulhat, hogy a szövegszerkesztőhöz tartozó szövegmemóriának csak egy része látszik a képernyőn. Így természetesen az is előfordulhat, hogy a szövegmemória adott pontján található szövegjel a szövegszerkesztő kép-

ernyőelem látható területén kívül található. Ez a függvény az ilyen szövegeleket a látható területre mozgatja.

A függvény első paramétere a szövegszerkesztőt, második paramétere pedig a szövegszerkesztőhöz tartozó szövegmemória egy szövegjelét jelöli a memóriában.

A függvény visszatérési értéke **TRUE**, ha a szövegjel el kellett mozdtani, hogy a szöveg látható területére kerüljön, **FALSE**, ha a szövegjel a függvény hívása előtt is a látható területen belül volt.

```
void gtk_text_view_scroll_mark_onscreen(GtkTextView
    *szövegszerkesztő, GtkTextMark *szövegjel);
```

A függvény működése nagyon hasonlít a `gtk_text_view_move_mark_onscreen()` függvény működésével, de azzal ellentétben nem a szövegjel, hanem a szöveget mozgatja. Ez a függvény a szövegszerkesztő képernyőelem mozgására szolgáló vízszintes és függőleges görgetősávok segítségével a szöveget helyzetét úgy állítja be, hogy a szövegjellel jelölt pont a képernyőn látható legyen.

A függvény első paramétere a szövegszerkesztőt, második paramétere a szövegjelét jelöli a memóriában. A visszatérési érték meghatározza, hogy szükség volt-e a szöveg görgetésére.

8.2.4. A cetlik

Amint azt már említettük, a cetlik a szövegmemória egyes részeihez rendelhetők, hogy így a szöveg adott részének megjelenését módosítsák. A cetlik segítségével a szöveg betűtípusát, betűméretét és betűváltozatát, a színét és más tulajdonságait is megváltoztathatjuk. Ha viszont a teljes szövegszerkesztő képernyőelem betűtípusát meg akarjuk változtatni, akkor szerencsésebb, ha nem címkét hozunk létre, hanem a 193. oldalon található 34. példának megfelelő módon járunk el.

A cetlik kezelésére használatos legfontosabb függvények a következők.

```
GtkTextTag *gtk_text_buffer_create_tag(GtkTextBuffer
    *szmemória, const gchar *cetlinév, const gchar *név1,
    érték1, const gchar *név2, érték2, ..., NULL);
```

A függvény segítségével új címkét hozhatunk létre a szövegmemória számára. A cetli a szöveg megjelenésének több tulajdonságát is befolyásolhatja. A függvény paramétereinek és visszatérési értékének jelentése a következő:

visszatérési érték A függvény visszatérési értéke a létrehozott címkét jelöli a memóriában.

Ha a cetli létrehozásakor nevet is megadtunk, a cetlire ezzel a névvel is hivatkozhatunk, nincs feltétlenül szükségünk a visszatérési érték tárolására.

szmemória A szövegmemóriát jelöli a memóriában, amelynek számára az adott címkét létrehozzuk. A cetli ennek a szövegmemóriának a szövegrészeihez rendelhető a létrehozása után.

cetlinév A cetli neve. Ha ennek a paraméternek az értéke egy karakterláncot jelöl a memóriában, a címkére a későbbiekben ezzel a névvel is hivatkozhatunk, ami roppant kényelmes használatot tesz lehetővé.

Ha ennek a paraméternek az értéke **NULL**, a címkére névvel nem hivatkozhatunk, csak a visszatérési értéként kapott mutatóval.

név1 Az szöveg első befolyásolt tulajdonságának neve. A használható tulajdonságokra a későbbiekben részletesen kitérünk.

érték1 A befolyásolt tulajdonság értéke, amelynek típusa a tulajdonság nevétől függ. A függvénynek tetszőleges számú név/érték párt átadhatunk, így a cetli a szövegrész több tulajdonságát is befolyásolhatja.

A név/érték tulajdonságpárok felsorolása után az utolsó paraméternek **NULL** értékűnek kell lennie, így kell jelölnünk a lista végét.

```
void gtk_text_buffer_insert_with_tags(GtkTextBuffer
    *szmemória, GtkTextIter *bejáró, const gchar *szöveg,
    gint hossz, GtkTextTag *cetli1, GtkTextTag *cetli2,
    ..., NULL);
```

A függvény segítségével új szövegrészt helyezhetünk el a szövegmemóriában úgy, hogy az új szövegrészre érvényes cetliket is meghatározhatjuk. A függvény működése és használata hasonlít a 212. oldalon bemutatott `gtk_text_buffer_insert()` függvény működéséhez és használatához, azzal a különbséggel, hogy ez a függvény a cetliket is figyelembe veszi.

A függvénynek paraméterként megadhatjuk azoknak a cetliknek a helyét a memóriában, amelyeket az új szövegrészre érvényesíteni szeretnénk. A listát egy **NULL** értékkel kell lezárnunk, ami a függvény utolsó paramétereként áll.

```
void gtk_text_buffer_insert_with_tags_by_name(GtkTextBuffer
    *szmemória, GtkTextIter *bejáró, const gchar *szöveg,
    gint hossz, const gchar *név1, const gchar *név2, ...,
    NULL);
```

A függvény segítségével új szövegrészt helyezhetünk el a szövegmemóriában úgy, hogy az új szövegrészre érvényes cetlik nevét is megadjuk. A függvény működése és használata hasonlít

a 212. oldalon bemutatott `gtk_text_buffer_insert()` függvény működéséhez és használatához, azzal a különbséggel, hogy ez a függvény a cetliket is figyelembe veszi.

A függvénynek a paraméterként megadhatjuk azoknak a cetliknek a nevét, amelyet az új szövegrészre érvényesíteni akarunk. A nevek listáját egy `NULL` értékkel kell lezárnunk, ami a függvény utolsó paramétereként áll.

```
void gtk_text_buffer_apply_tag(GtkTextBuffer *szmemória,
    GtkTextTag *cetli, const GtkTextIter *eleje, const
    GtkTextIter *vége);
```

A függvény segítségével a szövegmémória adott szövegrészhöz rendelhetünk címkét.

A függvény első paramétere a szövegmémóriát, a második paramétere pedig az érvényesíteni kívánt címkét jelöli a memóriában. A harmadik és negyedik paraméterek azokat a szövegbejárókat jelölik a memóriában, amelyek meghatározzák, hogy a címkét a szöveg mely részén kívánjuk alkalmazni.

```
void gtk_text_buffer_apply_tag_by_name(GtkTextBuffer
    *szmemória, const gchar *név, const GtkTextIter
    *eleje, const GtkTextIter *vége);
```

A függvény segítségével a szövegmémória adott szövegrészhöz rendelhetünk címkét, mégpedig a címkére nevével hivatkozva.

A függvény első paramétere a szövegmémóriát jelöli a memóriában. A második paraméter az érvényesíteni kívánt cetli nevét jelöli a memóriában. A harmadik és negyedik paraméterek azokat a szövegbejárókat jelölik a memóriában, amelyek meghatározzák, hogy a címkét a szöveg mely részén kívánjuk alkalmazni.

```
void gtk_text_buffer_remove_tag(GtkTextBuffer *szmemória,
    GtkTextTag *cetli, const GtkTextIter *eleje, const
    GtkTextIter *vége);
```

A függvény segítségével a szövegmémória adott szövegrészén a cetli hatását megszüntethetjük.

A függvény paramétereinek értelmezése megegyezik a `gtk_text_buffer_apply_tag()` függvény paramétereinek értelmezésével.

```
void gtk_text_buffer_remove_tag_by_name(GtkTextBuffer
    *szmemória, const gchar *név, const GtkTextIter
    *eleje, const GtkTextIter *vége);
```

A függvény segítségével a szövegmémória adott szövegrészén a cetli hatását megszüntethetjük, mégpedig úgy, hogy a címkére a nevével hivatkozunk.

A függvény paramétereinek értelmezése megegyezik a `gtk_text_buffer_apply_tag_by_name()` függvény paramétereinek értelmezésével.

`void gtk_text_buffer_remove_all_tags(GtkTextBuffer *szmemória, const GtkTextIter *eleje, const GtkTextIter *vége);` A függvény segítségével a szövegmemória adott szövegrészén az összes cetli hatását megszüntethetjük, hogy a szöveg a szövegszerkesztő alapértelmezett betűtípusával jelenjen meg.

A függvény első paramétere a szövegmemóriát jelöli a memóriában, második és harmadik paramétere pedig azokat szövegbejárókat jelöli a memóriában, amelyek a módosítani kívánt szövegrészlet elejét és végét jelölik.

Mielőtt rátérnénk a cetlik által befolyásolt szövegtulajdonságok nevének és lehetséges értékeinek részletes ismertetésére, egy példaprogram segítségével bemutatjuk a leggyakrabban használt szövegformázó eszközöket.

43. példa. A következő programrészlet bemutatja a szerkesztő létrehozásakor meghívott függvényt, ami a szövegszerkesztőben megjelenő szövegmemória számára létrehoz néhány címkét, majd feltölti a szövegmemóriát olyan tartalommal, ami bemutatja a cetlik hatását. A program hatására megjelenő képernyőelemet a 210. oldalon található 8.4. ábrán láthatjuk.

```

1 void
2 on_textview_realize(GtkWidget *widget,
3                      gpointer user_data)
4 {
5     GtkTextBuffer *text_buffer;
6     GtkTextIter iter;
7
8     text_buffer = gtk_text_view_get_buffer(
9         GTK_TEXT_VIEW(widget));
10
11     /*
12      * Előtérshíneket meghatározó cetlik.
13      */
14     gtk_text_buffer_create_tag(text_buffer, "blue",
15                               "foreground", "blue", NULL);
16     gtk_text_buffer_create_tag(text_buffer, "red",
17                               "foreground", "red", NULL);
18     gtk_text_buffer_create_tag(text_buffer, "yellowish",

```

```

19     "foreground", "#ffaa44", NULL);
20     /*
21     * Háttérszíneket meghatározó cetlik.
22     */
23     gtk_text_buffer_create_tag(text_buffer, "blue-back",
24     "foreground", "white",
25     "background", "blue", NULL);
26     gtk_text_buffer_create_tag(text_buffer, "red-back",
27     "foreground", "white",
28     "background", "red", NULL);
29     gtk_text_buffer_create_tag(text_buffer,
30     "yellowish-back",
31     "foreground", "white",
32     "background", "#ffaa44", NULL);
33     /*
34     * Betűcsaládokat meghatározó cetlik.
35     */
36     gtk_text_buffer_create_tag(text_buffer,
37     "family-sans", "family", "Sans", NULL);
38     gtk_text_buffer_create_tag(text_buffer,
39     "family-helv", "family", "Helvetica", NULL);
40     gtk_text_buffer_create_tag(text_buffer,
41     "family-times", "family", "Times", NULL);
42     gtk_text_buffer_create_tag(text_buffer,
43     "family-mono", "family", "Monospace", NULL);
44     /*
45     * Pontos betűtípust meghatározó cetlik.
46     */
47     gtk_text_buffer_create_tag(text_buffer, "complex",
48     "font", "Times Italic 16", NULL);
49     /*
50     * Aláhúzásokat meghatározó cetlik.
51     */
52     gtk_text_buffer_create_tag(text_buffer,
53     "underline-single",
54     "underline", PANGO_UNDERLINE_SINGLE, NULL);
55     gtk_text_buffer_create_tag(text_buffer,
56     "underline-double",
57     "underline", PANGO_UNDERLINE_DOUBLE, NULL);
58     gtk_text_buffer_create_tag(text_buffer,
59     "underline-low",
60     "underline", PANGO_UNDERLINE_LOW, NULL);
61     gtk_text_buffer_create_tag(text_buffer,
62     "underline-error",

```



```

63     "underline", PANGO_UNDERLINE_ERROR, NULL);
64
65     /*
66     * A cetlik hatását bemutató szövegrészek elhelyezése.
67     */
68     gtk_text_buffer_get_end_iter(text_buffer, &iter);
69
70     gtk_text_buffer_insert_with_tags_by_name(text_buffer,
71     &iter, "Kék betűszín, ", -1,
72     "blue", NULL);
73     gtk_text_buffer_insert_with_tags_by_name(text_buffer,
74     &iter, "vörös betűszín, ", -1,
75     "red", NULL);
76     gtk_text_buffer_insert_with_tags_by_name(text_buffer,
77     &iter, "sárgás betűszín...\n", -1,
78     "yellowish", NULL);
79
80     gtk_text_buffer_insert_with_tags_by_name(text_buffer,
81     &iter, "Kék háttérszín, ", -1,
82     "blue-back", NULL);
83     gtk_text_buffer_insert_with_tags_by_name(text_buffer,
84     &iter, "vörös háttérszín, ", -1,
85     "red-back", NULL);
86     gtk_text_buffer_insert_with_tags_by_name(text_buffer,
87     &iter, "sárgás háttérszín...\n", -1,
88     "yellowish-back", NULL);
89
90     gtk_text_buffer_insert_with_tags_by_name(text_buffer,
91     &iter, "Kék írógépbetűk.\n", -1,
92     "blue", "family-mono", NULL);
93
94     gtk_text_buffer_insert_with_tags_by_name(text_buffer,
95     &iter, "Előírt betűtípus.\n", -1,
96     "complex", NULL);
97
98     gtk_text_buffer_insert_with_tags_by_name(text_buffer,
99     &iter, "Egyszeres aláhúzás, ", -1,
100    "underline-single", NULL);
101    gtk_text_buffer_insert_with_tags_by_name(text_buffer,
102    &iter, "dupla, ", -1,
103    "underline-double", NULL);
104    gtk_text_buffer_insert_with_tags_by_name(text_buffer,
105    &iter, "alacsony, ", -1,
106    "underline-low", NULL);

```

```

107     gtk_text_buffer_insert_with_tags_by_name(text_buffer,
108         &iter, "hibajelző.\n", -1,
109         "underline-error", NULL);
110 }

```

A függvény a 14–63. sorok közt hozza létre a megjelenítést befolyásoló cetliket, a 70–109. sorok közt pedig elhelyezi a szövegmemóriában azokat a szövegrészeket, amelyek bemutatják a cetlik hatását.

A következő lista a cetlik lehetséges tulajdonságait sorolja fel. A listában első helyen a tulajdonság nevét, második helyen a tulajdonság típusát olvashatjuk. A név és a tulajdonságnak megfelelő érték a `gtk_text_buffer_create_tag()` függvény segítségével a cetli létrehozásakor megadható, vagy később a cetli tulajdonságaként a `g_object_set()` függvénnyel beállítható, illetve a `g_object_get()` függvénnyel lekérdezhető.

Megfigyelhető, hogy az egyes tulajdonságok mellett mindig megtalálható a tulajdonság érvényességét meghatározó logikai érték is. A **"background"** béven például beállíthatjuk a háttérszín, de mellette megtalálható a **"background-set"** is, amivel ki-, illetve bekapcsolhatjuk a háttérszín tulajdonság figyelembe vételét az adott cetli esetében. Az ilyen logikai értéket hordozó tulajdonságok alapértelmezett értéke a hamis, ami automatikusan igazra változik, amint beállítjuk a hozzájuk tartozó tulajdonságot.

"background", `gchararray` A szöveg háttérszínének neve karakterlánc-ként megadva. A név megadásakor használhatjuk a HTML nyelvben szokásos, 16-os számrendszert használót formát is (például **#13acf0**).

Az alapértelmezett érték a `NULL`, azaz a háttérszín a szövegszerkesztő képernyőelem háttérszínének megfelelő.

"background-full-height", `gboolean` Logikai érték, ami megadja, hogy a szöveg hátterének színe a teljes sormagasságban megjelenjen-e, vagy csak a karakter magasságáig terjedjen a hatása.

Az alapértelmezett érték a `FALSE`.

"background-full-height-set", `gboolean` Logikai érték, amely megadja, hogy az előző tulajdonság kifejtse-e a hatását.

Az alapértelmezett érték jelzi, hogy az előző tulajdonságot beállítottuk-e.

"background-gdk", `GdkColor` A szöveg hátterének színe `GdkColor` típus segítségével megadva.

238

"background-set", **gboolean** Logikai érték, ami megadja, hogy a háttérszín beállítás kifejtse-e a hatását.

Az alapértelmezett érték jelzi, hogy a háttérszínt beállítottuk-e.

"background-stipple", **GdkPixmap** A szöveg képernyőre rajzolásakor a háttér mintája, a szöveg mögött megjelenő kép.

Alapértelmezés szerint a háttér egyszínű.

"background-stipple-set", **gboolean** Logikai érték, ami megadja, hogy a szöveg rajzolásakor a háttér mintáját figyelembe kell-e venni.

Alapértelmezés szerint ez a tulajdonság megadja, hogy a háttér mintát megadtuk-e.

"direction", **GtkTextDirection** A szöveg írásának és olvasásának iránya, amelynek lehetséges értékei a következők:

GTK_TEXT_DIR_NONE Ismeretlen irány, amellyel esetleg jelezhetjük, hogy a szöveget jobbról balra és balról jobbra sem szerencsés elolvasni.

GTK_TEXT_DIR_LTR A szöveg írása és olvasása balról jobbra történik.

GTK_TEXT_DIR RTL Jobbról balra haladó szöveg.

Az alapértelmezett érték a **GTK_TEXT_DIR_NONE**.

"editable", **gboolean** Logikai érték, amely megadja, hogy a szövegnek ez a része módosítható-e.

Az alapértelmezett érték a **TRUE**.

"editable-set", **gboolean** Logikai érték, ami megadja, hogy az előző tulajdonságot figyelembe kell-e venni.

Alapértelmezett esetben ez a tulajdonság megadja, hogy az **"editable"** tulajdonságot beállítottuk-e.

"family", **GCharArray** A betűcsalád neve karakterláncként megadva. Használhatjuk például a Sans, Helvetica, Times, Monospace neveket.

A tulajdonság alapértelmezett értéke a **NULL**.

"family-set", **gboolean** Logikai érték, ami megadja, hogy a betűcsalád nevét figyelembe kell-e venni.

Az alapértelmezett érték megadja, hogy a betűcsaládot megadtuk-e.

"font", **gchararray** A használandó betűtípus neve – például „Sans Italic 12” –, ahogyan a betűtípus-ablak (lásd a 191. oldalon) lekérdezésekor azt megkaphatjuk.

Az alapértelmezett érték a **NULL**.

"font-desc", **PangoFontDescription** A betűtípus a Pango programkönyvtár adatszerkezetének segítségével megadva.

"foreground", **gchararray** A szöveg előtérshíze, azaz a betűk híze karakterláncként, a névvel megadva.

"foreground-gdk", **GdkColor** A szöveg híze **GdkColor** típus segítségével megadva.

"foreground-set", **gboolean** Logikai érték, ami megadja, hogy a szöveg hízeit figyelembe kell-e venni.

Az alapértelmezett érték megadja, hogy a betűk hízeit megadtuk-e.

"foreground-stipple", **GdkPixmap** A betűk rajzolásakor maszkként használandó kép.

"foreground-stipple-set", **gboolean** Logikai érték, ami megadja, hogy a betűk rajzolásakor a maszkt használunk-e.

Az alapértelmezett érték megadja, hogy az előző tulajdonságot beállítottuk-e.

"indent", **gint** A behúzás mértéke, ami megadja, hogy a bekezdés – a szövegmemória sora – első sorát hány képpontnyi értékkel kell beljebb kezdeni, mint a többi sort. Ez a tulajdonság lehet negatív is, ami jelzi, hogy az első sort a lapsélhez közelebb kell kezdeni.

A behúzást a szövegszerkesztő képernyőelem tulajdonságai közt a Glade programban is be lehet állítani, így, ha minden bekezdésnél azonos méretű behúzást akarunk használni, akkor esetleg ott érdemes a beállítást elvégezni.

Az alapértelmezett érték 0.

"indent-set", **gboolean** Logikai érték, ami meghatározza, hogy a beállított behúzást figyelembe kell-e venni.

Az alapértelmezett érték megadja, hogy a behúzást beállítottuk-e.

"invisible", **gboolean** Logikai érték, ha igaz, az adott szövegrész rejtett, nem látszik a képernyőn.

240

"invisible-set", **gboolean** Logikai érték, ami meghatározza, hogy az előző tulajdonságot figyelembe kell-e venni.

Alapértelmezett állapotban ez a tulajdonság megadja, hogy a szöveg láthatóságát megadtuk-e.

"justification", **GtkJustification** A szöveg rendezése, ami a következők egyike lehet:

GTK_JUSTIFY_LEFT A szöveg balra rendezett.

GTK_JUSTIFY_RIGHT A szöveg jobbra rendezett.

GTK_JUSTIFY_CENTER A szöveg középre rendezett.

GTK_JUSTIFY_FILL A szöveg sorkizárt.

A rendezést a szövegszerkesztő képernyőelem tulajdonságaként a Glade programban is beállíthatjuk.

Az alapértelmezett érték a **GTK_JUSTIFY_LEFT**.

"justification-set", **gboolean** Logikai érték, ami megadja, hogy a beállított rendezést figyelembe kell-e venni.

Az alapértelmezett érték megadja, hogy a rendezést beállítottuk-e.

"language", **gchararray** A szöveg nyelvének szabványos kódja karakterláncként.

Az alapértelmezett érték a **NULL**.

"language-set", **gboolean** Logikai érték, ami megadja, hogy a szöveg nyelvét figyelembe kell-e venni.

Az alapértelmezett érték megadja, hogy a szöveg nyelvét figyelembe vettük-e.

"left-margin", **gint** A bal oldali margó az üresen hagyandó képpontok számával megadva. Ez a tulajdonság nem lehet negatív. Tudnunk kell, hogy a bal oldali margó a szövegszerkesztő képernyőelem tulajdonságaként is beállítható a Glade programmal, így ha mindennél azonos bal margót akarunk használni, szerencsésebb ott elvégezni a beállítást.

Az alapértelmezett érték a 0.

"left-margin-set", **gboolean** Logikai érték, ami meghatározza, hogy a bal oldali margó beállítását figyelembe kell-e venni.

Az alapértelmezett érték megadja, hogy a margót beállítottuk-e.

"name", **gchararray** A cetli neve vagy **NULL**, ha a cetlit név nélkül hoztuk létre.

"paragraph-background", **gchararray** A bekezdéshez – a szövegmemória sorához – tartozó háttér színének neve.

Az alapértelmezett érték a **NULL**.

"paragraph-background-gdk", **GdkColor** A bekezdéshez tartozó háttérszín **GdkColor** típusú adatszerkezetként.

"paragraph-background-set", **gboolean** Logikai érték, ami megadja, hogy a bekezdéshez tartozó háttérszínt figyelembe kell-e venni.

Az alapértelmezett érték megadja, hogy a bekezdés háttérének színét beállítottuk-e.

"pixels-above-lines", **gint** Szám jellegű tulajdonság, aminek a segítségével megadhatjuk, hogy a szöveg bekezdései – a szövegmemória sorai – felett hány képpontnyi üres helyet kell kihagyni. Ennek a tulajdonságnak az értéke nem lehet negatív.

A szöveg bekezdései felett kihagyandó hely méretét a szövegszerkesztő tulajdonságai közt is beállíthatjuk a Glade segítségével, így ha minden bekezdés felett ugyanakkora helyet akarunk kihagyni, a beállítás ott is elvégezhető.

Az alapértelmezett érték 0.

"pixels-above-lines-set", **gboolean** Logikai érték, ami meghatározza, hogy az előző tulajdonságot figyelembe kell-e venni.

Az alapértelmezett érték megadja, hogy a bekezdések felett kihagyandó terület méretét megadtuk-e.

"pixels-below-lines", **gint** Szám jellegű tulajdonság, ami megadja, hogy a szöveg bekezdései – a szövegmemória sorai – alatt hány képpontnyi helyet kell kihagyni. Ez a tulajdonság a bekezdések felett kihagyandó hely méretéhez hasonlóan nem lehet negatív.

A bekezdések alatt kihagyandó terület magasságát a szövegszerkesztő képernyőelem tulajdonságai közt a Glade programban is be lehet állítani.

Az alapértelmezett érték 0.

"pixels-below-lines-set", **gboolean** Logikai érték, ami megadja, hogy az előző tulajdonságot figyelembe kell-e venni.

Az alapértelmezett érték megadja, hogy a bekezdés alatt kihagyandó üres hely méretét megadtuk-e.

"pixels-inside-wrap", **gint** Szám jellegű érték, ami megadja, hogy a szövegszerkesztő képernyőelem sorai közt hány képpont magasságú

üres helyet kell kihagynunk, mekkora legyen a sorköz. E tulajdonság nem lehet negatív értékű.

A sorok közt kihagyandó üres hely magasságát a szövegszerkesztő képernyőelemek tulajdonságai közt is megtalálható és a Glade programmal be is állíthatjuk. Ha a szöveg minden részére érvényes azonos értéket akarunk használni, akkor a beállítást érdemes lehet ott elvégezni.

Az alapértelmezett érték a 0.

"pixels-inside-wrap-set", gboolean Logikai érték, ami megadja, hogy a sortáv beállítást figyelembe kell-e venni.

Az alapértelmezett érték megadja, hogy a sortávot beállítottuk-e.

"right-margin", gint Szám jellegű tulajdonság, ami megadja a jobb oldalon kihagyandó üres terület szélességét – a jobb margót – képpontban mérve. Ez a tulajdonság nem lehet negatív.

A Glade segítségével a szövegszerkesztő képernyőelem hasonló tulajdonsága beállítható.

Az alapértelmezett érték 0.

"right-margin-set", gboolean Logikai érték, ami megadja, hogy a jobb oldali margó méretének beállítását figyelembe kell-e venni.

Az alapértelmezett érték megadja, hogy a jobb margót beállítottuk-e.

"rise", gint E szám jellegű jellemző segítségével beállíthatjuk a felső index mértékét, vagyis azt, hogy a szöveg betűinek alapvonala hány képpontnyival legyen magasabban a normális szöveg alapvonalánál.

E tulajdonság a Pango programkönyvtár belső mértékegységét használja, amelyet meg kell a **PANGO_SCALE** értékével, hogy a képpontok számát kapjuk.

Az alapértelmezett érték 0.

"rise-set", gboolean Logikai érték, ami megadja, hogy az előző tulajdonságot figyelembe kell-e vennünk.

Az alapértelmezett érték megadja, hogy a felső index mértékét beállítottuk-e.

"scale", gdouble Szám jellegű érték, ami megadja, hogy a betűméretet milyen mértékben kell növelnünk, illetve csökkentenünk. Ez az érték relatív, a téma által beállított betűméretet módosítja, így az 1,2 például azt jelenti, hogy a szöveget az alapértelmezett méretnél 1,2-szeresre növelt betűmérettel kell írunk.

A Pango programkönyvtár néhány szám jellegű állandót hoz létre a könnyebb használat és az egységes stílus érdekében. Ezek az állandók növekvő sorrendben a következők: `PANGO_SCALE_XX_SMALL`, `PANGO_SCALE_X_SMALL`, `PANGO_SCALE_SMALL`, `PANGO_SCALE_MEDIUM`, `PANGO_SCALE_LARGE`, `PANGO_SCALE_X_LARGE`, `PANGO_SCALE_XX_LARGE`.

Az alapértelmezett érték 1, ami nem módosítja a betűméretet.

"scale-set", **gboolean** Logikai érték, ami megadja, hogy az előző tulajdonságot figyelembe kell-e venni.

Az alapértelmezett érték megadja, hogy a betűméretet módosító értéket megadtuk-e.

"size", **gint** Szám jellegű érték, ami megadja, hogy hány képpont nagyságú betűket kell használnunk. A **"scale"** tulajdonsággal ellentétben ez a tulajdonság abszolút, nem az alapértelmezett betűmérethez képest határozza meg a cetli által előírt betűméretet.

Ez a tulajdonság a Pango programkönyvtár belső mértékegységében értendő, a `PANGO_SCALE` állandóval megszorozva alakíthatjuk a képpontok számát megadó értékre.

Az alapértelmezett érték 0, de alapértelmezett esetben – amikor a tulajdonságot nem állítottuk be – e tulajdonság sem módosítja a szöveg megjelenítését.

"size-points", **gdouble** A használandó betűméretet pont mértékegységben megadó szám jellegű érték.

Az alapértelmezett érték 0.

"size-set", **gboolean** Logikai érték, ami megadja, hogy a betűméret abszolút értékének beállítását figyelembe kell-e venni.

Alapértelmezett esetben meghatározza, hogy a betűméret abszolút értékét meghatároztuk-e.

"stretch", **PangoStretch** A betűk vízszintes nyújtását meghatározó tulajdonság. A Pango programkönyvtár a következő állandókat hozza létre a vízszintes nyújtás értékének meghatározására: `PANGO_STRETCH_ULTRA_CONDENSED`, `PANGO_STRETCH_EXTRA_CONDENSED`, `PANGO_STRETCH_CONDENSED`, `PANGO_STRETCH_SEMI_CONDENSED`, `PANGO_STRETCH_NORMAL`, `PANGO_STRETCH_SEMI_EXPANDED`, `PANGO_STRETCH_EXPANDED`, `PANGO_STRETCH_EXTRA_EXPANDED`, `PANGO_STRETCH_ULTRA_EXPANDED`.

"stretch-set", **gboolean** Logikai érték, ami megadja, hogy a vízszintes nyújtás értékét figyelembe kell-e venni.

244

Alapértelmezett esetben megadja, hogy a vízszintes nyújtást beállítottuk-e.

"strickethrough", **gboolean** Logikai érték, ami megadja, hogy a szöveget át kell-e húzni egy vízszintes vonallal.

Az alapértelmezett érték természetesen **FALSE**.

"strickethrough-set", **gboolean** Logikai érték, ami megadja, hogy a szöveg kihúzását figyelembe kell-e venni.

Alapértelmezett esetben megadja, hogy a kihúzást beállítottuk-e.

"style", **PangoStyle** A betűstílust megadó tulajdonság. A Pango programkönyvtár a következő **PangoStyle** típusú állandókat hozza létre a betűstílus meghatározására: **PANGO_STYLE_NORMAL**, **PANGO_STYLE_OBLIQUE**, **PANGO_STYLE_ITALIC**.

Az alapértelmezett érték a **PANGO_STYLE_NORMAL**.

"style-set", **gboolean** Logikai érték, ami megadja, hogy a betűstílust figyelembe kell-e venni.

Alapértelmezett esetben megadja, hogy a betűstílust beállítottuk-e.

"tabs", **PangoTabArray** A tabulátorpozíciókat beállító tulajdonság.

A Pango programkönyvtár a tabulátorpozíciók beállítására a **PangoTabArray** adatszerkezetben tárolja. Ilyen adatszerkezetet a legkönnyebben a **pango_tab_array_new_with_positions()** függvény segítségével hozhatunk létre.

"tabs-set", **gboolean** Logikai érték, ami megadja, hogy a **"tabs"** tulajdonság alapján megadott tabulátorpozíciókat figyelembe kell-e venni.

Alapértelmezett esetben megadja, hogy a tabulátorpozíciókat beállítottuk-e.

"underline", **PangoUnderline** A szöveg aláhúzását meghatározó tulajdonság. A Pango programkönyvtár a következő **PangoUnderline** típusú állandókat hozza létre az aláhúzás beállítására: **PANGO_UNDERLINE_NONE**, **PANGO_UNDERLINE_SINGLE**, **PANGO_UNDERLINE_DOUBLE**, **PANGO_UNDERLINE_LOW**, **PANGO_UNDERLINE_ERROR**.

"underline-set", **gboolean** Logikai érték, ami megadja, hogy a beállított aláhúzást figyelembe kell-e venni.

Alapértelmezett esetben megadja, hogy az aláhúzás típusát beállítottuk-e.

"variant", PangoVariant A kiskapitális betűváltozat beállítására szolgáló tulajdonság. A Pango programkönyvtár a következő **PangoVariant** típusú állandók segítségével teszi lehetővé a kiskapitális változat beállítását: **PANGO_VARIANT_NORMAL**, **PANGO_VARIANT_SMALL_CAPS**.

"variant-set", gboolean Logikai érték, ami megadja, hogy az előző tulajdonságot figyelembe kell-e venni.

Alapértelmezett esetben megadja, hogy a kiskapitális változatot meghatározó **"variant"** tulajdonságot beállítottuk-e.

"weight", gint Szám jellegű érték a betűk „vastagságának” meghatározására. Minél magasabb értéket képvisel ez a tulajdonság, a betűk annál robusztusabbak lesznek. A Pango programkönyvtár a következő szám jellegű állandókat hozza létre a betűk vastagságának meghatározására (a növekvő vastagság sorrendjében): **PANGO_WEIGHT_ULTRALIGHT**, **PANGO_WEIGHT_LIGHT**, **PANGO_WEIGHT_NORMAL**, **PANGO_WEIGHT_SEMI_BOLD**, **PANGO_WEIGHT_BOLD**, **PANGO_WEIGHT_ULTRABOLD**, **PANGO_WEIGHT_HEAVY**.

Az alapértelmezett érték a **PANGO_WEIGHT_NORMAL** ami 400-as számértéknek felel meg.

"weight-set", gboolean Logikai érték, ami meghatározza, hogy az előző tulajdonságot figyelembe kell-e venni.

Alapértelmezett esetben megadja, hogy a betűvastagságot meghatározó tulajdonságot beállítottuk-e.

"wrap-mode", GtkWrapMode A szöveg tördelését meghatározó tulajdonság. A tördelés módja meghatározza, hogy a szövegmemória (**GtkTextBuffer**) sorai milyen módon jelenjenek meg a képernyőn, a szövegszerkesztő képernyőelemben (**GtkTextView**).

A GTK programkönyvtár a következő **GtkWrapMode** típusú állandókat hozza létre a tördelés meghatározására:

GTK_WRAP_NONE A tördelés tiltása. Ha ezt az üzemmódot használjuk, a szövegmemória sorai a képernyőn egy sorban jelennek meg, a szövegszerkesztő képernyőelem olyan szélessé válik, hogy a leghosszabb sorok is kiférjenek. A Glade a szövegszerkesztő képernyőelemhez vízszintes görgetősávot is létrehoz, így segítve a hosszú sorok megjelenítését.

GTK_WRAP_CHAR Tördelés karakterhatáron. Ilyen üzemmódban a szövegszerkesztő a szövegmemória sorait a megjelenítéshez teljes szólegyes ponton tördelheti, akár szó belsejében is. A tördelés

csak a megjelenítést érinti – azaz a szövegszerkesztő nem helyez el újsor karaktereket a szövegben – azonban így is megnehezítheti az olvasását, hogy a szavak eldarabolva jelennek meg.

GTK_WRAP_WORD A tördelés a szóhatárok figyelembe vételével. Ebben az üzemmódban a szövegszerkesztő a szövegmemória tartalmát úgy jeleníti meg, hogy szükség esetén a szóhatárokon új sort kezd. A szöveg így olvasható marad, hiszen a szavak nem darabolódnak, szótöredékek nem kerülnek át a következő sorba.

Ebben az üzemmódban a szövegmemória sorai a képernyőn nekezdésekhez hasonlóan viselkednek, a szövegszerkesztő szélességének megfelelően esetleg több sorban jelennek meg. Innen ered az, hogy a GTK programkönyvtár dokumentációjában a „sor” és a „bekezdés” fogalma kissé egybemosódik.

GTK_WRAP_WORD_CHAR Ebben a módban a tördelés szóhatáron, ha szükséges szavak belsejében történik.

A tördelés módját a Glade program segítségével az egész szövegre beállíthatjuk.

Az alapértelmezett érték a **GTK_WRAP_NONE**.

"wrap-mode-set", **gboolean** Logikai érték, ami megadja, hogy az előző tulajdonságot figyelembe kell-e venni.

Alapértelmezett esetben megadja, hogy az adott cetli számára a szöveg tördelését beállítottuk-e.

8.2.5. A kurzor és a kijelölt szöveg

A **GtkTextView** képernyőelem a megjelenített szövegmemóriában mindig létrehoz két **GtkTextMark** szövegjelet, amelyek fontosak lehetnek a programozó számára is. A **insert** nevű szövegjel a szövegkurzor helyét jelöli ha nincsen kijelölt szöveg, és a kijelölt szöveg elejét jelöli, ha a felhasználó az egérrel vagy a billentyűzettel kijelöl egy szövegrészt. Ez utóbbi esetben a **selection_bound** nevű szövegjel a kijelölt szöveg végére mutat.

A szövegkurzor és a kijelölt szöveg kezelésében hasznunkra lehetnek a következő függvények:

```
GtkTextMark *gtk_text_buffer_get_insert(GtkTextBuffer
    *szövegmemória);
```

Ez a függvény a szövegkurzor helyét jelző szövegjel lekérdezésére használható. Ugyanezt a hatást érhetjük el, ha a **gtk_text_buffer_get_mark()** függvény segítségével az **insert** nevű szövegjelet kérdezzük le.

A függvény paramétere a szövegjelet jelöli a memóriában, amelynek a szövegkurzort jelölő szövegjelét szeretnénk megkeresni a memóriában. A függvény visszatérési értéke a szövegjelet jelöli a memóriában.

```
GtkTextMark *gtk_text_buffer_get_selection_bound(GtkTextBuffer
*szövegmemória);
```

Ez a függvény az előző függvényhez hasonlóan használható, a kijelölt szöveg végét jelző szöveggel megkeresésére használható.

```
gboolean gtk_text_buffer_get_has_selection(GtkTextBuffer
*szövegmemória);
```

A függvény segítségével megállapíthatjuk, hogy a szövegmemóriában van-e a felhasználó által kijelölt szöveg.

A függvény paramétere a szövegmemóriát jelöli, visszatérési értéke pedig `TRUE`, ha a szövegben van kijelölt rész, illetve hamis, ha nincs.

```
gboolean gtk_text_buffer_get_selection_bounds(GtkTextBuffer
*szövegmemória, GtkTextIter *eleje, GtkTextIter
*vége);
```

A függvény segítségével két szövegbejárót állíthatunk be, hogy azok a kijelölt szöveg elejére és végére mutassanak.

A függvény első paramétere a szövegmemóriát jelöli a memóriában, második paramétere azt a területet, ahol a kijelölt szöveg elejére állítandó szövegbejárót elhelyeztük, harmadik paramétere pedig értelemszerűen azt a memóriaterületet, ahová a függvény a kijelölt szöveg végére mutató szövegmemóriát el fogja helyezni.

A függvény visszatérési értéke igaz logikai érték, ha legalább egy karakternyi kijelölt szöveg van a szövegmemóriában.

```
void gtk_text_buffer_select_range(GtkTextBuffer
*szövegmemória, const GtkTextIter *eleje, const
GtkTextIter *vége);
```

E függvény segítségével a szövegmemória adott részét kijelölhetnénk, éppen úgy, mintha a kijelölést a felhasználó végezte volna el az egérrel vagy a billentyűzettel.

A függvény első paramétere a szövegmemóriát, második paramétere a kijelölendő szöveg elejét jelző bejárót, harmadik paramétere pedig a kijelölendő szöveg végét jelző bejárót jelzi a memóriában.

A szövegmemória kezelése során hasznos lehet a következő jelzések ismerete.

changed A szövegmemória akkor küldi ezt a jelzést, amikor a szövegmemóriában található szöveg megváltozik.

A jelzés visszahívott függvényének típusa a következő:

```
void név(GtkTextBuffer *szövegmemória, gpointer
adat);
```

A függvény első paramétere a szövegmemóriát jelzi a memóriában, amelyik a jelzést küldte. A függvény második paramétere egy mutató, amelyet a függvény nyilvántartásba vételekor a programozó adhat meg.

mark-set A szövegmemória akkor küldi ezt a jelzést, amikor a szövegmemóriához tartozó valamelyik szövegelet a program elmozdítja a helyéről. Akkor, ha a szöveggel az eredeti helyén marad, de a szöveg megváltoztatása – új szövegrész beszúrása vagy szövegrészlet törlése – miatt a szöveggel a szöveggel együtt elmozdul, ez a jelzés nem jelenik meg.

A jelzéshez visszahívott függvényként a következő függvénytípus kapcsolható:

```
void név(GtkTextBuffer *szövegmemória, GtkTextIter
*szövegbejáró, GtkTextMark *szöveggel, gpointer
adat);
```

Ilyen visszahívott függvényt készíthetünk a szövegelek mozgásának megfigyelésére.

A függvény első paramétere a szövegmemóriát, második paramétere a szöveggel új helyét jelző szövegbejárót jelöli a memóriában. A függvény harmadik paramétere a szövegelet jelöli a memóriában. A GTK programkönyvtár a alapértelmezés szerint a szöveggel mozgatása után hívja a visszahívott függvényt, így az a függvény hívásakor már a megfelelő helyet jelöli a szövegben. A függvény utolsó paramétere a programozó által a visszahívott függvény nyilvántartásba vételekor megadott mutató.

44. példa. Előfordulhat, hogy szeretnénk visszahívott függvényt készíteni, amit a GTK programkönyvtár akkor hív meg, ha a felhasználó a szövegkurzort elmozdítja a helyéről. Ezt a feladatot megfelelő típusú visszahívott függvény nyilvántartásba vételével oldhatjuk meg. A visszahívott függvény bejegyzését célszerű a szövegszerkesztő képernyőelem létrehozásakor, a *realize* jelzéshez rendelt visszahívott függvényben elvégezni:

```
1 void
2 on_textview_realize(GtkWidget *widget,
3                       gpointer user_data)
4 {
```

```

5   GtkTextBuffer *buffer;
6
7   buffer = gtk_text_view_get_buffer(
8       GTK_TEXT_VIEW(widget));
9   g_signal_connect((gpointer) buffer, "mark-set",
10      G_CALLBACK(mark_set_callback), NULL);
11 }
```

A függvény előbb lekérdezi a szövegszerkesztő képernyőelemhez tartozó szövegmemória címét a 7–8. sorban, majd a szövegmemória **mark-set** jelzéséhez rendeli a **mark_set_callback** függvényt a 9–10. sorokban. Ez utóbbi függvényt ez után a GTK programkönyvtár minden esetben hívja, ha a szövegmemória valamelyik szövegjelét új helyre kell áttenni.

A szövegkurzor mozgását figyelni visszahívott függvényt a következő formában készíthetjük el:

```

1   void
2   mark_set_callback(GtkTextBuffer *textbuffer,
3                     GtkTextIter   *iter,
4                     GtkTextMark   *mark,
5                     gpointer       user_data)
6   {
7       gchar *name = gtk_text_mark_get_name(mark);
8
9       if (name == NULL || strcmp(name, "insert") != 0)
10          return;
11
12       g_message("%s(): insert moved", __func__);
13   }
```

A függvény a 7. sorban lekérdezi az éppen mozgatott szövegjel nevét, majd a 9. sorban megvizsgálja a nevet. Ha a nevet jelölő mutató értéke **NULL** – azaz a szövegjelnek nincsen neve – vagy a név nem egyezik meg a **insert** karakterlánccal, a függvény a 10. sorban visszatér.

A függvény 12. sorára már csak akkor kerül a vezérlés, ha az éppen mozgatott szövegjel neve **insert**, ami mindig a szövegkurzor helyét jelöli.

A vágólap az alkalmazások közti kapcsolattartás egyik fontos és igen kényelmes formája, ami nagymértékben megkönnyíti a felhasználó munkáját. Úgy is tekinthetünk a vágólap támogatására, mint a programozó elemi udvariasságára, amit a felhasználó joggal várhat el minden esetben.

Linux rendszereken a vágólapnak legalább két változatát szokás használni. Ha a felhasználó az alkalmazásban kijelöl egy szövegrészt vagy grafikus elemet, akkor azt minden különös előkészület nélkül beillesztheti a

250

középső egérgomb lenyomásával. Ettől függetlenül egy másik vágólapot is szokás használni, ami a *szerkesztés* menü megfelelő menüpontjain, vagy azok gyorsbillentyűin keresztül használható.

A szövegszerkesztő képernyőelem automatikusan kezeli a középső egérgomb segítségével elérhető vágólapot, a másik, menüpontok segítségével elérhető vágólap azonban csak a programozó segítségével kezelhető.

A vágólapok nyilvántartására a GTK programkönyvtár a `GtkClipboard` típust hozza létre. Szerencsére a legtöbb alkalmazásban elegendő a következő függvény ismerete a vágólap lekérdezésére:

```
GtkClipboard *gtk_clipboard_get(GdkAtom atom);
```

A függvény segítségével az igényinknek megfelelő vágólapot címét kérdezhetjük le. A függvény paramétereként meg kell adnunk, hogy melyik vágólap címét szeretnénk visszatérési értéként megkapni. A vágólap használatához elegendő, ha a következő két állandó egyikét használjuk paraméterként:

GDK_SELECTION_PRIMARY Ennek az állandónak a hatására az egyszerűbb, a középső egérgomb használatán alapuló vágólap címét kapjuk vissza. Ezt a vágólapot a szövegszerkesztő képernyőelem automatikusan kezeli.

GDK_SELECTION_CLIPBOARD Ennek a változónak a hatására a szokásos, a *szerkesztés* menü használatán alapuló vágólap címét kapjuk vissza.

```
gboolean gtk_text_buffer_delete_selection(GtkTextBuffer
*szövegmemória, gboolean interaktív, gboolean
szerkeszthető);
```

A függvény segítségével a kijelölt szöveget törölhetjük. A törlés nem érinti a vágólap tartalmát, így ezt a függvényt általában a *szerkesztés* menü *törlés* menüpontjának visszahívott függvényében hívjuk.

A függvény visszatérési értékének és paramétereinek jelentése a következő:

visszatérési érték A függvény visszatérési érték igaz, ha a törlés sikeres volt.

szövegmemória Az első paraméter a szövegmemóriát jelöli, amelyben a módosítást el akarjuk végezni.

interaktív Ez a paraméter logikai érték, ami jelzi, hogy a műveletet a felhasználó közvetlen kérésére kell-e elvégezni. Ha ennek a paraméternek az értéke igaz, akkor a függvény meg fogja vizsgálni, hogy a kijelölt szöveg módosítható-e, ha hamis, akkor nem. Ennek az az oka, hogy a nem módosítható szöveget

a felhasználó nem módosíthatja, maga az alkalmazás azonban igen.

szerkeszthető Ez a paraméter megadja, hogy a szövegmemóriát alapértelmezett esetben – ha más nem módosítja – szerkeszthetőnek kell lennie. Ettől a függvénytől függetlenül a kijelölt szövegrész lehet szerkeszthető vagy védett is, hiszen ezt a tulajdonságot a cetlik segítségével is beállíthatjuk.

```
void gtk_text_buffer_paste_clipboard(GtkTextBuffer
    *szövegmemória, GtkClipboard *vágólap, GtkTextIter
    *hová, gboolean szerkeszthető);
```

A függvény segítségével a vágólap tartalmát a szövegbe illeszthetjük. Ezt a függvényt általában a **szerkesztés** menü **beillesztés** menüpontjának visszahívott függvényében hívjuk.

A függvény argumentumainak jelentése a következő:

szövegmemória Annak a szövegmemóriának a címe, ahová a vágólap tartalmát be akarjuk illeszteni.

vágólap A vágólap címe, amelynek tartalmát a szövegmemóriába be akarjuk illeszteni.

hová A szövegbejáró címe, ami kijelöli azt a helyet a szövegben, ahová a vágólap tartalmát be kívánjuk illeszteni. Ez a paraméter lehet **NULL** értékű is, ami jelzi, hogy a szöveget a szokásos helyre, a szövegkurzorhoz illesztjük be.

szerkeszthető Logikai érték, ami megadja, hogy a szövegmemóriát alapértelmezés szerint módosíthatónak tekintjük-e. A szöveg adott része ennek a paraméternek az értékétől függetlenül is lehet módosítható is és védett is, hiszen ezt a tulajdonságot a cetlik segítségével adott szövegrészekre is beállíthatjuk.

```
void gtk_text_buffer_copy_clipboard(GtkTextBuffer
    *szövegmemória, GtkClipboard *vágólap);
```

A függvény segítségével a szövegmemória kijelölt szövegét a vágólapra másolhatjuk. Ezt a függvényt általában a **szerkesztés** menü **másolás** menüpontjának visszahívott függvényében használjuk.

A függvény első paramétere a szövegmemóriát, másodikk pedig a használni kívánt vágólapot jelöli a memóriában.

```
void gtk_text_buffer_cut_clipboard(GtkTextBuffer
    *szövegmemória, GtkClipboard *vágólap, gboolean
    szerkeszthető);
```

A függvény segítségével a szövegmemória kijelölt szövegét a vágólapra másolhatjuk és a szöveget ezzel egyidőben törölhetjük. Ezt

252

a függvényt általában a *szerkesztés* menü *kivág* menüpontjának visszahívott függvényében használjuk.

A függvény első paramétere a szövegmemóriát, második paramétere pedig a vágólapot jelöli. A függvény harmadik paramétere megadja, hogy a szövegmemóriát alapértelmezés szerint módosíthatónak tekintjük-e.

A következő függvény a szövegmemóriát kezelő vágólap használatát mutatja be.

45. példa. A következő függvények a *szerkesztés* menü szokásos menüpontjainak visszahívott függvényeiként a vágólap kezelését végzik szövegszerkesztő képernyőelem és a hozzá tartozó szövegmemória támogatására.

```

1  void
2  on_cut1_activate(GtkMenuItem *menuitem,
3                   gpointer      user_data)
4  {
5      GtkWidget *widget;
6      GtkTextBuffer *text_buffer;
7      GtkClipboard *clipboard;
8
9      widget = lookup_widget(GTK_WIDGET(menuitem),
10                             "textview");
11      text_buffer = gtk_text_view_get_buffer(
12          GTK_TEXT_VIEW(widget));
13      clipboard = gtk_clipboard_get(
14          GDK_SELECTION_CLIPBOARD);
15
16      gtk_text_buffer_cut_clipboard(text_buffer, clipboard,
17                                   TRUE);
18  }
19
20
21  void
22  on_copy1_activate(GtkMenuItem *menuitem,
23                   gpointer      user_data)
24  {
25      GtkWidget *widget;
26      GtkTextBuffer *text_buffer;
27      GtkClipboard *clipboard;
28
29      widget = lookup_widget(GTK_WIDGET(menuitem),
30                             "textview");

```

```

31     text_buffer = gtk_text_view_get_buffer(
32         GTK_TEXT_VIEW(widget));
33     clipboard = gtk_clipboard_get(
34         GDK_SELECTION_CLIPBOARD);
35
36     gtk_text_buffer_copy_clipboard(text_buffer,
37         clipboard);
38 }
39
40
41 void
42 on_paste1_activate(GtkMenuItem *menuitem,
43                    gpointer      user_data)
44 {
45     GtkWidget *widget;
46     GtkTextBuffer *text_buffer;
47     GtkClipboard *clipboard;
48
49     widget = lookup_widget(GTK_WIDGET(menuitem),
50         "textview");
51     text_buffer = gtk_text_view_get_buffer(
52         GTK_TEXT_VIEW(widget));
53     clipboard = gtk_clipboard_get(
54         GDK_SELECTION_CLIPBOARD);
55
56     gtk_text_buffer_paste_clipboard(text_buffer,
57         clipboard, NULL, TRUE);
58 }
59
60
61 void
62 on_delete1_activate(GtkMenuItem *menuitem,
63                     gpointer      user_data)
64 {
65     GtkWidget *widget;
66     GtkTextBuffer *text_buffer;
67
68     widget = lookup_widget(GTK_WIDGET(menuitem),
69         "textview");
70     text_buffer = gtk_text_view_get_buffer(
71         GTK_TEXT_VIEW(widget));
72
73     gtk_text_buffer_delete_selection(text_buffer, TRUE,
74         TRUE);

```

254

75 | }

A programrészletben megfigyelhetjük a *szerkesztés* menü *kivág* (1–18. sorok), *másol* (21–38. sorok), *beilleszt* (41–58. sorok) és *töröl* (61–75. sorok) menüpontjainak visszahívott függvényeit.

8.2.6. A szöveg módosításának figyelése

A szövegszerkesztő képernyőelemben szerkesztett szöveget általában állományban tároljuk, az állomány mentésének, betöltésének és a programból való kilépésnek a megszervezése közben pedig általában tudnunk kell, hogy a szöveg a betöltés óta módosult-e.

A módosítás figyelésére saját eszközöket is készíthetünk, sokkal hatékonyabb és gyorsabb azonban az alábbi két függvény használata:

```
void gtk_text_buffer_set_modified(GtkTextBuffer
    *szövegmemória, gboolean módosult);
```

A függvény segítségével beállíthatjuk, hogy a szöveg módosult-e. Általában arra használjuk ezt a függvényt, hogy minden mentés végén beállítsuk a módosultság értékét hamisra. A szövegmemória automatikusan „módosult” állapotba kapcsolódik, amikor a felhasználó megváltoztatja a szövegét.

A függvény első paramétere a szövegmemóriát jelöli, a második paramétere pedig megadja a „módosult” jellemző új értékét, ami `TRUE`, illetve `FALSE` lehet.

```
gboolean gtk_text_buffer_get_modified(GtkTextBuffer
    *szövegmemória);
```

A függvény segítségével lekérdezhetjük a szövegmemória „módosult” jellemzőjét.

8.3. A raktárak

A GTK+ néhány összetett képernyőelem – ikonmező, fa stb. – kezeléséhez raktárakat (*store*, raktár) használ. A raktárakban adatokat helyezhetünk el, amelyeket az adott képernyőelem megjelenít, kezel. Ha ilyen összetett képernyőelemeket akarunk használni a programunkban, létre kell hoznunk és kezelnünk kell a megfelelő raktárat.

8.3.1. Az általános raktár

A GTK programkönyvtár a `GtkListStore` típussal lista szerkezetű raktárat, a `GtkTreeStore` típussal pedig fa szerkezetű raktárat hoz létre a

programozó számára. Mindkét típus megvalósítja a `GtkTreeModel` interfészt, így mindkét típus átalakítható `GtkTreeModel`, általános raktár típusú a `GTK_TREE_MODEL()` típuskényszerítő makró segítségével. Ez rendkívül fontos, mert az adatokat megjelenítő képernyőelemek `GtkTreeModel` típust fogadnak el az adatok átadásakor.

A GTK programkönyvtár a `GtkTreeModel` típushoz bejáróként a `GtkTreeIter` típust biztosítja, amellyel így a `GtkListStore` és a `GtkTreeStore` típusú raktárakat is bejárhatjuk. Hasonlóképpen használhatók a lista és fa szerkezetű raktárak elemeinek bejárására a `GtkTreePath` típusú ösvényleíró adatszerkezetek is. A bejárókat általában helyi változóként rövid élettartammal, az ösvényleírókat pedig a legtöbb esetben hosszabb élettartamú eszközként kiterjedtebb hatókörrel használjuk (hasonlóan a `GtkTextIter` és `GtkTextMark` adatszerkezetekhez).

A raktárak kezelésekor sokszor használunk szöveges ösvényleírókat, ami az állományleíróhoz hasonlóan, szöveges formában határozza meg a raktár egy bizonyos pontját. A szöveges ösvényleíró kettőspontokkal elválasztott egész számok sorozata. A `"1:2"` például a raktár 1 sorszámu sorának 2 sorszámu leszármazottját jelöli a fa szerkezetű raktárban. Az elemek számozása 0-tól indul, így a `"0:0:0"` például a legelső sor legelső leszármazottjának legelső leszármazottja.

A GTK programkönyvtár a következő függvényeket biztosítja a `GtkTreeIter` típusú fabejárók kezelésére:

```
gboolean gtk_tree_model_get_iter_from_string(GtkTreeModel
    *raktár, GtkTreeIter *bejáró, const gchar
    *ösvényleíró);
```

A függvény segítségével a szöveges ösvényleíróból állíthatunk elő fabejárót.

A függvény első paramétere a raktárat, második paramétere pedig azt a területet jelöli a memóriában, ahová a függvény a fabejárót elhelyezi. Ez utóbbi paraméternek olyan területet kell jelölnie a memóriában, amely elegendően nagyméretű egy fabejáró tárolására. A függvény harmadik paramétere egy szöveges ösvényleírókat jelöl (például `"1:0:3"`), ami alapján a fabejáró elkészül.

A függvény visszatérési értéke `TRUE`, ha az adott ösvényleíró alapján a fabejárót el lehetett készíteni, egyébként pedig `FALSE`.

```
gboolean gtk_tree_model_get_iter_first(GtkTreeModel
    *raktár, GtkTreeIter *bejáró);
```

A függvény segítségével a bejáró legelső elemét jelölő fabejárót állíthatjuk elő.

A függvény első paramétere a raktárat, második paramétere pedig a bejáró elhelyezésére kijelölt memóriaterületet jelöli. A függvény visszatérési értéke **TRUE**, ha a bejárót sikerült kitölteni, **FALSE**, ha nem. Ez utóbbi kizárólag akkor következhet be, ha a raktárban egyetlen elem sem található.

```
gboolean gtk_tree_model_iter_next(GtkTreeModel *raktár,
    GtkTreeIter *bejáró);
```

A függvény segítségével a bejárót a következő, azonos szinten található elemre állíthatjuk. A függvény a bejárót nem az általa jelölt faelemnek leszármazottjára, hanem a következő elemre állítja.

A függvény első paramétere a raktárat jelöli a memóriában. A második paraméter azt a bejárót jelöli, amelyet a következő elemre szeretnénk átállítani.

A függvény visszatérési értéke **TRUE**, ha az adott elem után volt következő elem, különben pedig **FALSE**. Ez utóbbi esetben a függvény a bejárót érvénytelen értékűre állítja.

```
gboolean gtk_tree_model_iter_children(GtkTreeModel
    *raktár, GtkTreeIter *bejáró, GtkTreeIter *szülő);
```

E függvény segítségével az adott bejáró által jelölt faelem első leszármazottját kereshetjük ki.

A függvény első paramétere a raktárat jelöli a memóriában. A második paraméter azt a bejárót jelöli, amelyet a harmadik paraméter által jelölt bejáró első gyermekére akarunk állítani. Ha a függvény harmadik paraméterének értéke **NULL**, a függvény a második paraméterrel jelölt bejárót a raktár legelső elemére állítja. Ez különösen a faelemeket bejáró ciklus elején lehet hasznos.

A függvény visszatérési értéke **TRUE**, ha a bejárót sikerült beállítani, **FALSE**, ha nem.

```
gboolean gtk_tree_model_iter_has_child(GtkTreeModel
    *raktár, GtkTreeIter *bejáró);
```

A függvény segítségével lekérdezhetjük, hogy az adott elemnek van-e leszármazottja.

A függvény első paramétere a raktárat, a második eleme a fabejárót jelöli a memóriában. A visszatérési érték **TRUE**, ha a második paraméter által jelölt bejárónak van leszármazottja, **FALSE**, ha nincs.

```
gint gtk_tree_model_iter_n_children(GtkTreeModel *raktár,
    GtkTreeIter *bejáró);
```

A függvény segítségével lekérdezhetjük, hogy az adott raktárelemnek hány leszármazottja van.

A függvény első paramétere a raktárat, második eleme a fabejárót jelöli a memóriában. Ha ez utóbbi paraméter értéke `NULL`, a függvény a raktár gyökerében található elemek számát adja vissza.

A visszatérési érték megadja a raktár adott helyén található elem leszármazottjainak számát.

```
gboolean gtk_tree_model_iter_nth_child(GtkTreeModel
    *raktár, GtkTreeIter *bejáró, GtkTreeIter *szülő, gint
    n);
```

A függvény segítségével a bejárót az adott elem adott sorszámu gyermekére állíthatjuk.

A függvény első paramétere a raktárat, második paramétere a beállítandó bejárót jelöli a memóriában. A függvény harmadik paramétere a bejárót jelöli, amelynek az adott sorszámu gyermekét keressük. Ha ez a paraméter `NULL`, a függvény az adott sorszámu gyökérelmét keresi meg. A negyedik paraméter a keresett gyermek sorszámu adja meg. Az elemek számozása 0-tól kezdődik.

A függvény visszatérési értéke `TRUE`, ha a keresett sorszámu gyermek megtalálható volt, `FALSE`, ha nem. Ebben az esetben a függvény a második paramétere által jelölt bejárót érvénytelen értékre állítja.

```
gboolean gtk_tree_model_iter_parent(GtkTreeModel *raktár,
    GtkTreeIter *bejáró, GtkTreeIter *gyermek);
```

A függvény segítségével az adott elem szülőelemét kereshetjük ki.

A függvény első paramétere a raktárat, második paramétere a beállítandó bejárót jelöl a memóriában. A függvény harmadik paramétere egy bejárót jelöl a memóriában, ami azt az elemet jelöli, amelynek szülőjét keressük.

A függvény visszatérési értéke `TRUE`, ha a harmadik paraméter által jelölt elemnek van szülője, `FALSE`, ha az elem gyökérelm. Ez utóbbi esetben a függvény a második paramétere által jelölt bejárót érvénytelen értékre állítja.

```
gchar *gtk_tree_model_get_string_from_iter(GtkTreeModel
    *raktár, GtkTreeIter *bejáró);
```

A függvény segítségével az adott bejáró által jelölt elem szöveges ösvényleíróját kaphatjuk meg.

A függvény első paramétere a raktárat, második eleme pedig a bejárót jelöli a memóriában.

A függvény visszatérési értéke egy dinamikusan foglalt memóriaterületet jelöl a memóriában, ami a szöveges ösvényleírókat tartalmazza karakterláncként.

A `GtkTreePath` típusú ösvényleírók a `GtkTreeIter` fabejárókhhoz hasonlóan a raktár egy elemét azonosítják, de az ösvényleírók használata közben nem szabad szem előtt téveszteniük, hogy azok nem garantálják, hogy az adott elem létezik a raktárban. Ha például a megfelelő függvénnyel az ösvényleírókat a következő elemre állítjuk lehetséges, hogy olyan ösvényleírókat kapunk, ami nem jelöl elemet a raktárban, mégpedig egyszerűen azért, mert nincsen következő elem. Az ösvényleírókat kezelő függvények tehát az adott műveletet akkor is elvégzik, ha az eredményül kapott ösvényleíró a művelet után érvénytelen helyre mutat.

A GTK programkönyvtár a következő függvények segítségével támogatja a `GtkTreePath` típusú ösvényleírók használatát:

`GtkTreePath *gtk_tree_model_get_path(GtkTreeModel *raktár, GtkTreeIter *bejáró);` A függvény segítségével ösvényleírókat állíthatunk elő az általa jelölt elem megadásával.

A függvény első paramétere a raktárat jelöli a memóriában. A második paramétere azt a fabejárót jelöli, ami meghatározza, hogy a létrehozott ösvényleíró a raktár melyik elemét jelölje.

A függvény visszatérési értéke a dinamikusan foglalt memóriaterületen létrehozott fabejárót jelöli a memóriában. Ezt az adatszerkezetet a `gtk_tree_path_free()` függvény segítségével semmisíthetjük meg.

`gboolean gtk_tree_model_get_iter(GtkTreeModel *raktár, GtkTreeIter *bejáró, GtkTreePath *ösvény);` A függvény segítségével a fabejárót az ösvény által meghatározott elemre állíthatjuk.

A függvény első paramétere a raktárat, második paramétere pedig a beállítandó bejárót jelöli a memóriába, míg a harmadik paraméter az elemet jelölő ösvény címe.

A függvény visszatérési értéke `TRUE`, ha az elemet sikerült megtalálni és így a bejáró beállítása lehetséges volt, ellenkező esetben viszont `FALSE`.

`GtkTreePath *gtk_tree_path_new(void);` Ennek a függvénynek a segítségével egy új ösvényleírókat hozhatunk létre. A függvény nem túl hasznos, hiszen nem tudjuk meghatározni, hogy az ösvény a raktár melyik elemét jelölje.

A függvény visszatérési értéke a dinamikusan foglalt memóriaterületen elhelyezett ösvényleírókat jelöli a memóriában.

GtkTreePath *gtk_tree_path_new_from_string(const gchar *ösvény); E függvény segítségével új ösvényleíró hozhatunk létre a szöveges ösvényleíró alapján.

A függvény paramétere a szöveges ösvényleíró jelöli a memóriában. A visszatérési érték a dinamikus memóriaterületre mutat, ahol a függvény az új ösvényleíró létrehozta.

gchar *gtk_tree_path_to_string(GtkTreePath *ösvény); A függvény segítségével szöveges ösvényleíró hozhatunk létre az ösvényleíró alapján.

A függvény első paramétere az ösvényleíró jelöli a memóriában, ami alapján a szöveges ösvényleíró létrehozuk. A függvény visszatérési értéke dinamikusan foglalt memóriaterületet jelöl, ahol a függvény a szöveges ösvényleíró elhelyezte a memóriában.

GtkTreePath *gtk_tree_path_new_first(void); A függvény a legelső elemet jelölő ösvényt állítja elő, hatása megegyezik a **gtk_tree_path_new_from_string()** függvény hatásával, ha annak a "0" szöveges ösvényleíró adjuk paraméterként.

void gtk_tree_path_append_index(GtkTreePath *ösvény, gint n); Ennek a függvénynek a segítségével az ösvényhez új elemet fűzhetünk, azaz elérhetjük, hogy az ösvény az addig jelölt elem *n*-edik gyermekét jelölje.

A függvény első paramétere az ösvényleíró jelöli a memóriában, a második paramétere pedig a hozzáfűzendő szám jellegű értéket adja meg. A második paraméter meghatározza, hogy az ösvény az eredeti érték hányadik gyermekét jelölje.

void gtk_tree_path_prepend_index(GtkTreePath *ösvény gint n); A függvény segítségével az ösvény elejére új értéket helyezhetünk el. Ez a függvényt nyilvánvalóan nem túl gyakran fogjuk használni.

gint gtk_tree_path_get_depth(GtkTreePath *ösvény); A függvény segítségével lekérdezhethetjük az ösvény mélységét.

A függvény paramétere az ösvényleíró jelöli a memóriában, a visszatérési értéke pedig megadja az ösvény mélységét.

GtkTreePath *gtk_tree_path_copy(const GtkTreePath *ösvény); A függvény segítségével egy új ösvényleíró hozhatunk létre a megadott ösvényleíró lemásolásával.

A függvény paramétere a mintaként használt ösvényt jelöli a memóriában, míg a visszatérési értéke a másolatként létrehozott új ösvényt, ami dinamikusan foglalt memóriaterületen van elhelyezve.

`gint gtk_tree_path_compare(const GtkTreePath *a, const GtkTreePath *b);` A függvény segítségével két ösvényleíró összehasonlíthatunk össze.

A függvény két paramétere a két összehasonlítandó ösvényleíró jelöli a memóriában, visszatérési értéke pedig a `strcmp()` szabványos könyvtári függvényhez hasonlóan negatív, 0 vagy pozitív értéket ad, annak megfelelően, hogy az első ösvényleíró kisebb, a két ösvényleíró egyenlő, vagy az első ösvényleíró nagyobb.

`void gtk_tree_path_next(GtkTreePath *ösvény);` A függvény segítségével az ösvényleíró a következő raktárelemre állíthatjuk. Ez a függvény az ösvényleíró az azonos mélységben található következő elemre állítja, nem növeli az ösvényleíró mélységét.

A függvény paramétere az átállítandó ösvényleíró jelöli a memóriában.

`gboolean gtk_tree_path_prev(GtkTreePath *ösvény);` A függvény segítségével az ösvényleíró az azonos szintén lévő előző elemre állíthatjuk, ha az ösvény nem az adott szinten található 0 sorszámú elemet jelöli.

A függvény paramétere az átállítandó ösvényleíró jelöli a memóriában, visszatérési értéke pedig megadja, hogy az ösvény átállítása sikeres volt-e.

`gboolean gtk_tree_path_up(GtkTreePath *ösvény);` A függvény segítségével az ösvényleíró egy szinttel feljebb, a gyökér felé mozgathatjuk, ha az ösvényleíró nem gyökérelemet jelölt.

A függvény paramétere az átállítandó ösvényt jelöli a memóriában, visszatérési értéke pedig jelzi, hogy az átállítás sikeres volt-e.

`void gtk_tree_path_down(GtkTreePath *ösvény);` A függvény segítségével az ösvényleíró mélységét eggyel növelhetjük, átállíthatjuk az eddig jelzett elem 0 sorszámú elemére.

A függvény paramétere az átállítandó ösvényleíró jelöli a memóriában.

`gboolean gtk_tree_path_is_ancestor(GtkTreePath *ösvény, GtkTreePath *leszármazott);` A függvény segítségével megállapíthatjuk, hogy az egyik ösvény a másik ösvény leszármazottját jelöli-e.

A függvény igaz logikai értéket ad, ha az első paraméterként jelzett ösvény a második paraméterként jelzett ösvény őse.

Az ösvényleírókat és fabejárókat kezelő függvények bemutatása után sorra vesszük a raktárak kezelésére használható függvényeket. A GTK programkönyvtár igen sok függvényt biztosít a `GtkTreeModel` típusú általános raktár kezelésére. Nem szabad azonban elfelejtenünk, hogy az általános raktár valójában egy lista vagy fa szerkezetű raktár, ezért nem találunk például általános raktárat létrehozó függvényt, létrehozni csak lista vagy fa szerkezetű raktárat tudunk.

`gint gtk_tree_model_get_n_columns(GtkTreeModel *raktár);` A függvény segítségével lekérdezhethetjük, hogy az általános raktárban hány oszlopba vannak rendezve az adatok.

`GType gtk_tree_model_get_column_type(GtkTreeModel *raktár, gint oszlop);` A függvény segítségével lekérdezhethetjük, hogy az általános raktár adott oszlopának milyen a típusa.

A függvény első paramétere a raktárat jelöli a memóriában, a második paramétere pedig megadja, hogy hányadik oszlop típusát akarjuk lekérdezni. Az oszlopok számozása 0-tól indul.

A függvény visszatérési értéke megadja az adott oszlop típusát. A visszatérési értéként kapott `GType` típusról bővebben a 263. oldalon olvashatunk.

`void gtk_tree_model_get(GtkTreeModel *raktár, GtkTreeIter *bejáró, oszlop1, mutató1, ..., -1);` Ennek a függvénynek a segítségével a raktár bejáróval megadott pontjáról olvashatjuk ki az egyes oszlopok tartalmát.

A függvény első paramétere a raktárat, második paramétere a sort kijelölő bejárót jelöli a memóriában.

A függvény további paramétere az olvasandó oszlop számát, valamint az olvasott értékek elhelyezési címeit megadó párok, amelyekből tetszőlegesen sokat megadhatunk. A függvény utolsó paramétere kötelezően `-1`, a paraméterlista végét jelzi.

`void gtk_tree_model_foreach(GtkTreeModel *raktár, GtkTreeModelForeachFunc függvény, gpointer adat);` Ennek a függvénynek a segítségével a raktár minden sorát, minden ágát bejárhatjuk, minden soron, minden ágon meghívva egy adott függvényt.

A függvény első paramétere a bejárandó raktárat, a második az elemeken meghívandó függvényt jelöli a memóriában. A függvény harmadik paramétere egy tetszőleges mutató, amelyet a második paraméter által jelzett függvénynek paraméterként át szeretnénk adni.

```
gboolean (*GtkTreeModelForeachFunc)(GtkTreeModel *raktár,  
GtkTreePath *ösvény, GtkTreeIter *bejáró, gpointer  
adat); A gtk_tree_model_foreach() függvénynek második  
paraméterként átadott mutató típusa, úgy is mondhatnánk, hogy  
ilyen típusú függvényt tudunk hívni az általános raktár automati-  
kus bejárása során. A függvény paraméterei és visszatérési értéke  
a következő:
```

visszatérési érték A függvénynek **TRUE** visszatérési értékkel kell jeleznie, ha a raktár további elemeit már nem kell bejárni, illetve **FALSE** értékkel, ha kész a következő elem feldolgozására.

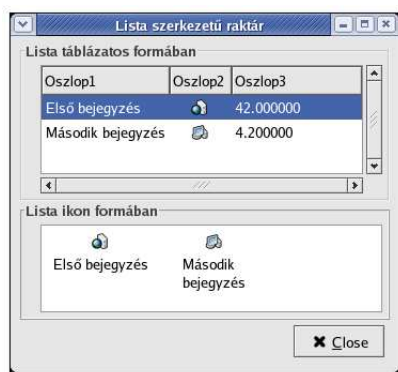
raktár Az éppen bejárt raktárat jelölő mutató.

ösvény Az aktuális elemet jelölő ösvény.

bejáró Az aktuális elemet jelölő bejáró.

adat A programozó által meghatározott mutató, amelyet a bejárás kezdeményezésekor a `gtk_tree_model_foreach()` függvény harmadik paramétereként megadott.

8.3.2. A lista szerkezetű raktár



8.5. ábra. Lista szerkezetű raktár kétféle nézetben

Amint arról az előző bekezdésekben olvashattunk, a lista szerkezetű raktár, azaz a `GtkListStore` típus segítségével különféle adatokat tárolhatunk lista formában. Ezt a lehetőséget általában arra használjuk, hogy a változó adatokat megjelenítő összetett képernyőelemeknek a megjelenítendő adatlistát adjuk.

A 8.5. ábrán egyszerű lista szerkezetű raktár két megjelenítési formáját láthatjuk. Felül listaként, a kép alsó részén pedig ikonlista formájában jelenítettük meg ugyanazokat az adatsorokat. A raktárban két sornyi adat található, az első oszlop szöveges, a második képet, a harmadik pedig egy lebegőpontos számot tartalmaz. Amint megfigyelhetjük az ikonmezőben csak az első két oszlop, a szöveges és az ikon jellegű látható.

A lista szerkezetű raktár – amelynek kezelésére a `GtkListStore` típust használhatjuk – tehát többoszlopos, különféle típusú adatrészeket tároló összetett adatszerkezet, ami leginkább egy táblázatra hasonlít.

A GTK programkönyvtár `GtkListStore` kezelésére létrehozott legegyszerűbb és legfontosabb függvényei a következők:

`GtkListStore *gtk_list_store_new(gint oszlopszám, ...);` A függvény segítségével egy új lista szerkezetű raktárat készíthetünk az oszlopok számának, és az egyes oszlopokban elhelyezendő adatok típusának megadásával.

A függvénynek megadott első paraméter a listában szereplő oszlopok számát adja meg.

A függvény további paraméterei a lista oszlopainak típusát határozzák meg. Ezeknek a paramétereknek a száma magától értetődően meg kell, hogy egyezzen a lista oszlopainak számával.

Az oszlopok típusát a G programkönyvtár `GType` típusának segítségével adhatjuk meg. A legfontosabb `GType` típusú állandók a következők:

`G_TYPE_STRING` Karakterlánc tárolására alkalmas típus. Akkor használjuk ezt a kulcsszót, ha az adott mezőben karakterláncot akarunk megjeleníteni, tárolni.

A GTK+ programkönyvtár a karakterláncról másolatot készít, amikor azt a lista típusú raktárban elhelyezzük és a másolat tárolására használt memóriaterületet felszabadítja, amikor az adott sort a listából eltávolítjuk.

`G_TYPE_BOOLEAN` Logikai érték tárolására alkalmas típus.

`G_TYPE_INT` Egész számok tárolására használható oszlopot létrehozó kulcsszó.

`G_TYPE_DOUBLE` Lebegőpontos számok tárolására alkalmas oszlopot létrehozó kulcsszó.

`G_TYPE_POINTER` Mutatók tárolására alkalmas oszlopot létrehozó kulcsszó.

`GDK_TYPE_PIXBUF` Kép, ikon tárolására alkalmas típus. Akkor használjuk ezt a kulcsszót, ha az adott oszlopban képet akarunk tárolni, megjeleníteni. (Ikonok létrehozására legegyszerűbben a `gdk_pixbuf_new_from_file()` függvény használható.)

A GTK – valójában a GDK programkönyvtár – nem készít másolatot az ilyen elemekről, de egy hivatkozási-szám nyilvántartás segítségével gondoskodik arról, hogy azok ne semmisüljenek meg, amíg a lista típusú raktárban szerepelnek.

A függvény visszatérési értéke a létrehozott lista típusú raktárat jelöli a memóriában.

`void gtk_list_store_clear(GtkListStore *raktár);`

A függvény segítségével a raktárból az összes elemet – összes sort – törölhetjük. A függvénynek átadott paraméter a raktárat jelöli a memóriában.

```
void gtk_list_store_append(GtkListStore *raktár,
    GtkTreeIter *bejáró);
```

A függvény segítségével a lista típusú raktárba új sort helyezhetünk el. Az új sor a lista végén, az utolsó létező elem után jön létre.

A függvénynek átadott első paraméter a raktárat jelöli a memóriában, ahová az új sort el akarjuk helyezni. A függvény második paramétere egy bejárót jelöl a memóriába, ahová a függvény az új elem kijelölésére, módosítására használható adatokat helyezi el. A függvény a második paraméterrel jelölt memóriaterületet módosítja, ezért annak egy `GtkTreeIter` típusú változót kell jelölnie.

Amint látjuk a függvény egy új sort helyez el a listában, de az új sorba adatokat nem tesz. Az adatokat a függvény által kitöltött bejáró segítségével egy külön függvény hívásával (az alább bemutatott `gtk_list_store_set()` függvénnyel) írhatjuk be az új sorba.

```
void gtk_list_store_prepend(GtkListStore *raktár,
    GtkTreeIter *bejáró);
```

A függvény működése és használata megegyezik a `gtk_list_store_append()` függvény működésével és használatával, azzal ellentétben azonban nem a lista végére, hanem a lista elejére helyezi el az új sort.

```
void gtk_list_store_set(GtkListStore *raktár, GtkTreeIter
    *bejáró, szám1, érték1, ..., -1);
```

A függvény segítségével a lista szerkezetű raktár adott sorában található adatokat módosíthatjuk. Általában ezt a függvényt használjuk arra, hogy a listában éppen elhelyezett új sort adatokkal feltöltsük.

A függvény első paramétere a lista szerkezetű raktárat jelöli a memóriában, második paramétere pedig a bejárót, amelyet a módosítani kívánt sor kiválasztására használunk.

A függvény további paraméterei az adott sorban elhelyezendő adatokat adják meg, mégpedig `oszlopszám, érték` formában. Az oszlopok számozása 0-tól indul, az értékek pedig értelemszerűen meg kell felelni a raktár adott oszlopának típusának.

Az oszlopszámokból és értékekből tetszőlegesen sokat megadhatunk, a lista végét azonban az oszlopszám helyén megadott `-1` értékkel kell jelölnünk. Ennek az utolsó paraméternek az elhagyása súlyos programhibához vezethet.

Már ezzel a néhány függvénnyel is készíthetünk lista szerkezetű raktárat, ahogyan ezt a következő példa is bemutatja.

46. példa. A következő programrészlet egy lista szerkezetű raktár létrehozását és adatokkal való feltöltését mutatja be. A függvény által létrehozott raktár kétféle megjelenési formáját láthatjuk a 8.5. ábrán.

```

1  static GtkListStore *
2  dull_list_store(void)
3  {
4      static GtkListStore *list_store = NULL;
5      GdkPixbuf *pixbuf1, *pixbuf2;
6      GtkTreeIter iter;
7
8      /*
9       * Ha már létrehoztuk a raktárat egyszerűen
10      * visszaadjuk.
11      */
12      if (list_store != NULL)
13          return list_store;
14
15      pixbuf1 = gdk_pixbuf_new_from_file(
16          PIXMAP_DIR "server.png",
17          NULL);
18      pixbuf2 = gdk_pixbuf_new_from_file(
19          PIXMAP_DIR "client.png",
20          NULL);
21
22      /*
23       * Új raktár három oszloppal.
24       */
25      list_store = gtk_list_store_new(3,
26          G_TYPE_STRING,
27          GDK_TYPE_PIXBUF,
28          G_TYPE_DOUBLE);
29
30      /*
31       * Két sornyi adatot elhelyezünk a raktárban.
32       */
33      gtk_list_store_append(list_store, &iter);
34      gtk_list_store_set(list_store, &iter,
35          0, "Első bejegyzés",
36          1, pixbuf1,
37          2, 42.0,
38          -1);
39
40      gtk_list_store_append(list_store, &iter);

```

```

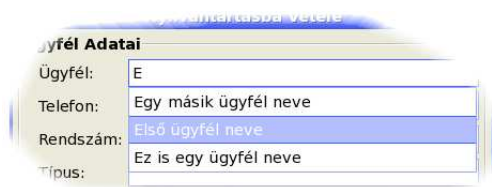
41     gtk_list_store_set(list_store, &iter,
42                         0, "Második bejegyzés",
43                         1, pixbuf2,
44                         2, 4.20,
45                         -1);
46
47     return list_store;
48 }
```

A példaként bemutatott függvény egy új lista szerkezetű raktárat hoz létre és feltölti adatokkal.

Magát a lista létrehozását a 25–28. sorokban olvashatjuk. Figyeljük meg, hogy a lista három oszlopot tartalmaz, amelyek közül az első karakterlánc, a második kép, a harmadik pedig szám jellegű.

A függvény a raktárban két sort helyez el. Az első adatsort a 33. sorban hívott `gtk_list_store_append()` függvény helyezi el a raktárban és a 34–38. sorban hívott `gtk_list_store_set()` tölti fel adatokkal.

A raktár második sorát a 40–45. sorok közt hozzuk létre és töltjük fel adatokkal.



8.6. ábra. Automatikus kiegészítés beviteli mezőhöz

A beviteli mezők felszerelhetők úgynevezett automatikus kiegészítéssel, amelyek segítik a felhasználót a szöveg beírásában. Az automatikus kiegészítés valójában egy lista, amelyből a GTK programkönyvtár a már beírt szöveg alapján az egyező szövegsorokat automatikusan megjeleníti, hogy a felhasználónak ne kelljen a teljes szöveget begépelnie. Az automatikus kiegészítésre példát a 8.6. ábrán láthatunk.

Az automatikus kiegészítés léterhozása és használata viszonylag egyszerű, ezért a szükséges függvényeket nem tárgyaljuk részletesen, csak egy példán keresztül mutatjuk be ket. A következő példa bemutatja hogyan készíthetünk automatikus kiegészítést beviteli mezőhöz egy SQL kiszolgálón található adatok alapján.

47. példa. A következő függvény egy SQL lekérdezést hajt végre, majd a lekérdezés során kapott adatokból egy lista szerkezetű raktárat hoz létre, hogy azt felhasználja az adott beviteli mezőhöz rendelt automatikus kiegészítés létrehozására. Az SQL kiszolgálóval való munkára e könyvben nem térünk részletesen, az ilyen célra használt függvényekről az olvasó más forrásból [1] tájékozódhat.

```

1 void
2 sql_create_entry_completion(
3     GtkEntry *entry,
4     PGconn *connection,
5     const gchar *sql_command)
6 {
7     GtkEntryCompletion *entry_completion;
8     GtkListStore *list_store;
9     GtkTreeIter tree_iter;
10    PGresult *result;
11    ExecStatusType status;
12    gint n, ntuples;
13
14    result = PQexec(connection, sql_command);
15    status = PQresultStatus(result);
16    if (status != PGRES_TUPLES_OK) {
17        g_warning("%s(): %s", __func__,
18            PQresultErrorMessage(result));
19        PQclear(result);
20        return;
21    }
22
23    list_store = gtk_list_store_new(1, G_TYPE_STRING);
24
25    ntuples = PQntuples(result);
26    for (n = 0; n < ntuples; ++n) {
27        gtk_list_store_append(list_store, &tree_iter);
28        gtk_list_store_set(list_store, &tree_iter,
29            0, PQgetvalue(result, n, 0),
30            -1);
31    }
32    PQclear(result);
33
34    entry_completion = gtk_entry_completion_new();
35    gtk_entry_completion_set_model(entry_completion,
36        GTK_TREE_MODEL(list_store));
37    gtk_entry_completion_set_text_column(
38        entry_completion, 0);
39    gtk_entry_set_completion(entry, entry_completion);
40 }

```

A függvény paraméterként a beviteli mező, az SQL kiszolgáló felé felépített kapcsolatot leíró adatszerkezet, valamint a használandó SQL parancsot

kijelölő mutatókat kap. A paraméterek ilyen módon történő megválasztása általános felhasználhatóságot kölcsönöz a függvénynek.

A függvény a 14. sorban az SQL parancsot elküldi a kiszolgálónak, majd a 15–20. sorokban megvizsgálja, hogy a parancs végrehajtása hibamentes volt-e. Ha hiba lépett fel, akkor a függvény a 17–18. sorokban egy figyelmeztetést ír ki az SQL programkönyvtár által adott hibaleírással (18. sor), hogy a programozót figyelmeztesse a problémára.

Ezek után a függvény a 23. sorban létrehoz egy lista szerkezetű raktárat egyetlen szöveges mezővel, majd egy ciklus segítségével a 26–30. sorok közt elhelyezi a raktárban a lekérdezés során kapott sorok első oszlopát. Az újabb és újabb elemek létrehozását a 27., a szöveges érték beállítását pedig a 28–30. sorok tartalmazzák.

A függvény ezek után a 32. sorban felszabadítja az SQL parancs eredményét tároló adatszerkezetet, hiszen a szövegsorok ekkor már a lista szerkezetű raktárban vannak.

A következő lépés az automatikus kiegészítés létrehozása és beállítása. A függvény a 34. sorban a `gtk_entry_completion_new()` függvény segítségével létrehozza az új automatikus kiegészítést, majd a 35–36. sorokban a `gtk_entry_completion_set_model()` függvény hívásával beállítja, hogy a kiegészítés az adatokat az éppen létrehozott lista szerkezetű raktárból vegye.

Következő lépésként a 37–38. sorban a `gtk_entry_completion_set_text_column()` függvény segítségével beállítjuk, hogy az automatikus kiegészítés a lista szerkezetű raktár melyik oszlopából történjen. Ez igen fontos lépés, semmiképpen nem szabad elfelejtenünk!

Az utolsó feladat az automatikus kiegészítés beviteli mezőhöz rendelése, amelyet a 39. sorban a `gtk_entry_set_completion()` hívásával végzünk el.

8.3.3. A fa szerkezetű raktár

Az általános raktár másik megjelenési formája a fa szerkezetű raktár, amelyben a lista szerkezetű raktárhoz hasonló módon több oszlopban tárolhatunk adatokat az egyes sorokhoz leszármazottakat is rendelve. A fa szerkezetű raktár kezelésére a GTK programkönyvtár a `GtkTreeStore` típust biztosítja a programozó számára.

A fa szerkezetű raktár kezelésére létrehozott függvények közül a legfontosabbak a következők:

```
GtkTreeStore *gtk_tree_store_new(gint oszlopszám, ...);
```

A függvény segítségével új fa szerkezetű raktárat hozhatunk létre.

A függvény első paramétere a létrehozandó raktár oszlopainak számát kell megadnunk, míg a további paraméterei az egyes oszlopok típusát adják meg. Mindig pontosan annyi típust kell megadnunk,

ahány oszlop létrehozását az első paraméter előír. A a függvény paramétereinek értelmezése megegyezik a `gtk_list_store_new()` függvény (263). oldal) paramétereinek értelmezésével.

A függvény visszatérési értéke az újonan létrehozott fa szerkezetű raktárat jelöli a memóriában.

`void gtk_tree_store_clear(GtkTreeStore *raktár);` A függvény segítségével a fa szerkezetű raktárból az összes elemet törölhetjük.

A függvény paramétere a fa szerkezetű raktárat jelöli a memóriában.

`void gtk_tree_store_append(GtkTreeStore *raktár, GtkTreeIter *bejáró, GtkTreeIter *szülő);` A függvény segítségével egy új sort helyezhetünk el a fa szerkezetű raktárban. Ez a függvény a fa megadott eleme leszármazottainak végére másol egy új sort, amelynek értékeit az alább bemutatott `gtk_list_store_set()` függvénnyel kell beállítanunk.

A lista és fa szerkezetű raktárak kezelésére használható függvények általában azonos paraméterezésűek, de ez a függvény különbözik a lista szerkezetű raktárban új elemet elhelyező `gtk_list_store_append()` függvénytől, hiszen ha a fában szeretnénk új elemet elhelyezni, akkor meg kell adnunk az új elem szülőjét is.

A függvény első paramétere a fa szerkezetű raktárat jelöli a memóriában. A második paraméter azt a bejárót jelöli, amelyet a függvény az általa létrehozott új elemre fog állítani, míg a harmadik paraméter annak a bejárónak a címét adja meg, amelyik kijelöli azt az elemet, amelynek az új elem az utolsó leszármazottja lesz. Ha ez utóbbi paraméter értéke `NULL`, akkor a függvény az új elemet a fa szerkeztű raktár gyökerében hozza létre.

`void gtk_tree_store_prepend(GtkTreeStore *raktár, GtkTreeIter *bejáró, GtkTreeIter *szülő);` A függvény segítségével a fa szerkezetű raktárban új elemet hozhatunk létre, mégpedig valamelyik elem első leszármazottjaként.

A függvény paramétereinek értelmezése és a működése is megegyezik a `gtk_tree_store_append()` függvélynél megfigyeltekkel, azaz a különbséggel, hogy ez a függvény nem a leszármazottak listájának végén, hanem az elején hozza létre az új elemet.

`void gtk_tree_store_set(GtkTreeStore *raktár, GtkTreeIter *bejáró, ..., -1);` A függvény segítségével a fa szerkezetű raktár adott eleme által hordozott adatokat állíthatjuk be. E függvénynek a használata megegyezik a lista szerkezetű raktár hasonló függvényének használatával.

A függvény első paramétere a raktárat, második paramétere pedig a módosítandó elemre állított bejárat jelöli a memóriában. A függvény további paraméterei a módosítandó oszlopok számából és a raktárban elhelyezendő új értékből álló párok. A paraméterlista végét a `-1` értékkel kell jeleznünk.

8.4. A cellarajzolók

A cellarajzolók a GTK programkönyvtár által biztosított, különféle adat-típusokat megjelenítő eszközök, amelyek az adatokat megjelenítő képernyőelem egyes részeihez rendelhetők a különféle adatok megjelenítésére. A 8.7. ábrán láthatjuk hogyan jelennek meg különféle cellarajzolók a képernyőn.



8.7. ábra. A cellarajzolók munka közben

Az ábrán két kombinált dobozt láthatunk. Mindkét dobozban egy kép és egy karakterlánc jelenik meg. Ezt a szerkezetet úgy hoztuk létre, hogy a kombinált dobozhoz két cellarajzoló rendeltünk, egy, a bal oldalon megjelenő képet készítő cellarajzoló és tőle balra egy szöveget megjelenítő cellarajzoló. A példa mindkét cellarajzója a megjelenítendő adatot egy lista szerkezetű raktár megfelelő ti-

pusú oszlopából veszi, így a kombinált dobozhoz tartozó lista minden sorában a lista szerkezetű raktár megfelelő sorában tárolt kép és szöveg jelenik meg.

A cellarajzolók, a fa és lista szerkezetű raktárak és az összetett adatszerkezetek megjelenítésére használható képernyőelemekből tehát olyan összetett viselkedésű rendszert készíthetünk, ami elég rugalmasak ahhoz, hogy az igényeinknek megfelelő formában tegyék lehetővé az adatkezelést a felhasználó számára. Ennek azonban ára is van, hiszen a cellarajzolók használata nem mindig egyszerű.

A GTK programkönyvtár a cellarajzolók alaptípusaként a `GtkCellRenderer` típust hozza létre. Ilyen típusú adatszerkezetet általában nem hozunk létre, konkrét cellarajzolóként általában a `GtkCellRenderer` leszármazottjaiként megvalósított – és a következő oldalakon bemutatott – típusokat használjuk. Úgy is gondolhatunk tehát a `GtkCellRenderer` típusra mint az objektumorientált programozás egy absztrakt osztályára, ami nem példányosítható, kizárólag az öröklődés segítségével használható.

A `GtkCellRenderer` a `GtkObject` leszármazottja, tehát nem képernyőelem, hiszen nem a `GtkWidget` osztály leszármazottjaként készült. A `GtkObject` viszont a `GObject` közvetett leszármazottja, így a `GtkCellRenderer` és leszármazottjai kezelhetők a G programkönyvtár

tulajdonságok tárolására használt `g_object_set()` és `g_object_get()` függvényeivel.

A G programkönyvtár által kezelt tulajdonságok – amelyeket a dokumentációban a *properties* cím alatt találhatunk – ráadásul igen fontos szerepet töltenek be a cellarajzolók működésében. Ennek az az oka, hogy a cellarajzoló képes arra, hogy a konkrét cella tulajdonságait lista vagy fa szerkezetű raktárból olvassa.

Ha például a lista képernyőelem első oszlopának a megjelenítésére szöveges cellarajzólót hozunk létre és a listához rendelt második oszlopban különféle színek angol nyelvű neveit helyezzük el, akkor megtehetjük, hogy a első oszlopban megjelenő cellarajzoló háttérszínt beállító tulajdonságát a lista szerkezetű raktár második oszlopához rendeljük és így elérjük, hogy a képernyőn minden sor a megfelelő háttérszínnel jelenjen meg.

A következő lista a `GtkCellRenderer` legfontosabb tulajdonságait tartalmazza. Ezeket a tulajdonságokat minden cellarajzoló esetében használhatjuk. A tulajdonságok közül néhány nagyon hasonlít a 237. oldalon bemutatott tulajdonságokhoz, ezért ezeket csak vázlatosan írjuk le.

"cell-background", `gchararray` A cellában használt háttérszín neve, vagy szöveges HTML kódja.

"cell-background-gdk", `GdkColor` A cellában használt háttérszín `GdkColor` típussal megadva.

"cell-background-set", `gboolean` Logikai érték, ami megadja, hogy a háttérszínt figyelembe kell-e venni a cella rajzolásakor.

"height", `gint` A cella magassága képpontban mérve. Ezt a tulajdonságot csak akkor kell megadnunk, ha nem akarjuk, hogy a GTK programkönyvtár automatikusan állítsa be a magasságot. Az alapértelmezett érték `-1`, ami jelzi, hogy a cella mindig a szükséges magasságú helyet foglalja el.

"is-expanded", `gboolean` Logikai érték, ami megadja, hogy az adott cella „nyitva” vagy „zárva” van-e. Ha a cellákat a képernyőn faként jelenítjük meg, akkor az egyes ágakat, alpontokat kinyithatjuk és becsukhatjuk ennek a tulajdonságnak a segítségével.

"sensitive", `gboolean` Logikai érték, ami megadja, hogy az adott cella érzékeny-e a felhasználó által keltett ingerekre.

"visible", `gboolean` Logikai érték, ami megadja, hogy a cella látható-e, esetleg rejtett.

"width", gint A cella szélessége képpontokban mérve. Ezt a tulajdonságot csak akkor kell beállítanunk, ha azt akarjuk, hogy a cella szélességét a GTK programkönyvtár ne automatikusan állítsa be. Ennek a tulajdonságnak az alapértelmezett értéke -1 , ami jelzi, hogy a cella szélessége automatikusan állítandó.

"xalign", gfloat FIXME: Nem működik?

"xpad", guint A cella bal oldalán üresen hagyandó terület szélessége képpontban mérve. Ennem a tulajdonságnak a segítségével egy oszlopon belül érzékeltethetjük az alárendeltségi viszonyokat, a faszerkezetet, ami igen hasznos lehet ha például kombinált dobozhoz tartozó listát készítünk.

"yalign", gfloat FIXME: Nem működik?

"ypad", guint A cella felső részén üresen hagyandó terület magassága képpontban mérve.

A cellarajzoló tulajdonságainak beállítására a későbbiekben, a konkrét cellarajzoló és képernyőelemek tárgyalása közben részletesen is bemutatjuk.

8.4.1. A szöveges cellarajzó

A szöveges cellarajzó az adatsorokat szöveges formában jeleníti meg a képernyőn mint ahogyan azt az egyszerű kombinált dobozban, egyszerű listákban is megfigyelhetjük. A GTK programkönyvtár a szöveges cellarajzó készítésére és használatára a **GtkCellRendererText** típust biztosítja a programozó számára.



8.8. ábra. A szöveges cellarajzó

A szöveges cellarajzó a szöveg egyszerű megjelenítésénél többre is képes. Ha például a szöveges cellarajzót használjuk egy listában a lehetőséget adhatunk a felhasználónak a szöveg átírására, befolyásolhatjuk a szöveg és a háttér színét, meghatározhatjuk a cellában megjelenő szöveg tördelését, azaz az egyszerű listánál összetettebb viselkedésű és megjelenésű képernyőelemet készíthetünk, ami jobban megfelel az igényeinknek. A szerkeszthető cellarajzó megjelenését mutatja be a 8.8. ábra, ahol a cellarajzót szerkesztés közben figyelhetjük meg. A szöveges cellarajzó létrehozására a következő függvényt használhatjuk:

```
GtkCellRenderer *gtk_cell_renderer_text_new(void);
```

A függvény segítségével új szöveges cellarajzót hozhatunk létre. A függvény visszatérési az új cellarajzót jelöli a memóriában. Ahhoz,

hogy a cellarajzoló használata is tudjuk azt egy cellák használatára felkészített képernyőelemhez kell rendelnünk.

A következő lista a szöveges cellarajzoló tulajdonságait mutatja be. A tulajdonságok közt vannak olyanok, amelyeket a cetlikkel kapcsolatban is használhatunk, és amelyeket emiatt már részletesen bemutatunk a 8.2.4. oldalon kezdődő listában. Ezeket a tulajdonságokat csak vázlatosan ismertetjük.

"attributes", **PangoAttrList** Ennek a tulajdonságnak a segítségével a szöveg összes jellemzőjét egyszerre adhatjuk meg. Erre a legtöbb esetben nincs szükség, hiszen a tulajdonságokat egyenként, külön-külön is beállíthatjuk.

Ezt a tulajdonságot **PangoAttrList** adatszerkezetként kell megadnunk, amelyet a **pango_attr_list_new()** függvény segítségével hozhatunk létre.

"background", **gchararray** A szöveg háttérszínének neve, vagy HTML kódja.

"background-gdk", **GdkColor** A szöveg hátterének színe **GdkColor** adatszerkezet segítségével megadva.

"background-set", **gboolean** Logikai érték, ami megadja, hogy a háttérszín figyelembe kell-e venni.

"editable", **gboolean** Logikai érték, ami megadja, hogy a szöveges cellarajzoló lehetővé teszi-e, hogy a felhasználó megváltoztassa, átírja a cellában megjelenő szöveget.

Ha ennek a tulajdonságnak az értéke igaz, a felhasználó a szövegre kattintva egy beviteli mezőt jeleníthet meg, amivel a szöveget a szokásos módon megváltoztathatja.

"editable-set", **gboolean** Logikai érték, ami meghatározza, hogy az előző tulajdonságot figyelembe kell-e venni.

"ellipsize", **PangoEllipsizeMode** Ez a tulajdonság meghatározza, hogy mit kell tenni, ha a cella rajzolásához rendelkezésre álló területre nem fér ki a szöveg. A tulajdonság a következő állandók egyikét veheti fel értékként:

PANGO_ELLIPSIZE_NONE Ennek az állandónak a hatására a Pango programkönyvtár semmilyen különleges lépést nem tesz, a szöveg vége egyszerűen nem fog megjelenni a képernyőn. Ez az alapértelmezett érték.

PANGO_ELLIPSIZE_START Ennek az állandónak a segítségével elérhetjük, hogy a Pango programkönyvtár a szöveg elejéről a megfelelő számú betűt eltávolítsa és ezt az eltávolított karakterek helyén megjelenített „...” karakterekkel jelezze.

PANGO_ELLIPSIZE_MIDDLE A szöveg rövidítése a szöveg közepének eltávolításával és a „...” jelek beírásával. Ez a módszer különösen az állományok elérési útjának rövidítésére használható, hiszen ott a szöveg eleje és vége fontosabb, mint a közepe.

PANGO_ELLIPSIZE_END A szöveg rövidítése a szöveg végének levágásával és a „...” karakterek elhelyezésével.

"ellipsize-set", **gboolean** Logikai érték, ami jelzi, hogy a rövidítés módját figyelembe kell-e venni.

"family", **gchararray** A használandó betűcsalád neve.

"family-set", **gboolean** Logikai érték, ami jelzi, hogy a beállított betűcsaládot figyelembe kell-e venni.

"font", **gchararray** A betűtípus teljes neve.

"font-desc", **PangoFontDescription** A betűtípus a Pango programkönyvtár adatszerkezeteként megadva.

"foreground", **gchararray** A előtérszín neve.

"foreground-gdk", **GdkColor** Az előtérszín a GDK programkönyvtár adatszerkezeteként megadva.

"foreground-set", **gboolean** Logikai típus, ami megadja, hogy az előtérszínt figyelembe kell-e venni.

"language", **gchararray** A használt nyelv szöveges kódja.

"language-set", **gboolean** Logikai érték, ami megadja, hogy a nyelvet figyelembe kell-e venni.

"markup", **gchararray** A megjelenítendő szöveg a Pango programkönyvtár jelölőnyelvét is figyelembe véve.

Ha a szöveges cellarajzolóban különféle egyszerű szövegtípusokat, színeket akarunk használni a szöveg kiemelésére, de nem akarunk az itt bemutatott tulajdonságok használatával vesződni, akkor jó ötlet lehet a Pango jelölőnyelv használata. Ha viszont azt akarjuk, hogy a cellarajzoló a szöveg kiírásakor figyelembe vegye a jelölőnyelv elemeit, akkor a szöveget a **"markup"** tulajdonsághoz rendelve kell megadnunk.

"rise", gint A szöveg alapvonalától mért távolsága.

"rise-set", gboolean Logikai érték, ami megadja, hogy az előző tulajdonságot figyelembe kell-e venni.

"scale", gdouble A betűméret nagyításának mérete.

"scale-set", gboolean Logikai érték, ami megadja, hogy a betűtípus nagyításának értékét figyelembe kell-e venni.

"single-paragraph-mode", gboolean Logikai érték, ami megadja, hogy a megjelenítendő szöveget mindenképpen egy bekezdésként kell-e megjeleníteni.

"size", gint A használandó betűméret.

"size-points", gdouble A használandó betűméret pont mértékegységben.

"size-set", gboolean Logikai érték, ami megadja, hogy a betűméretet figyelembe kell-e venni.

"stretch", PangoStretch A vízszintes nyújtás mértéke.

"stretch-set", gboolean Logikai érték, ami meghatározza, hogy a nyújtást figyelembe kell-e venni.

"strikethrough", gboolean Logikai érték, ami megadja, hogy a szöveget át kell-e húzni.

"strikethrough-set", gboolean Logikai érték, ami megadja, hogy az áthúzás beállítását figyelembe kell-e venni.

"style", PangoStyle A betűstílus.

"style-set", gboolean Logikai érték, ami megadja, hogy a beállított betűstílust figyelembe kell-e venni.

"text", gchararray A megjelenítendő szöveg karakterláncként megadva.

Ha a szöveges cellarajzoló segítségével olyan szöveget akarunk megjeleníteni, amelyben a Pango programkönyvtár jelölőnyelvét nem akarjuk figyelembe venni, akkor a megjelenítendő szöveget ehhez a tulajdonsághoz kell rendelnünk.

"underline", PangoUnderline Az aláhúzás típusa.

"underline-set", gboolean Logikai érték, ami megadja, hogy az aláhúzás módját figyelembe kell-e venni.

276

"variant", **PangoVariant** A betűváltozat.

"variant-set", **gboolean** Logikai érték, ami megadja, hogy a betűváltozatot figyelembe kell-e venni.

"weight", **gint** A betűvastagság.

"weight-set", **gboolean** Logikai érték, ami megadja, hogy a beállított betűvastagságot figyelembe kell-e venni.

"width-chars", **gint** A cella szélessége a megjeleníthető karakterek számában megadva. Az alapértelmezett érték -1, ami jelzi, hogy ezt a tulajdonságot nem kell figyelembe venni.

"wrap-mode", **PangoWrapMode** A tördelés módja.

"wrap-width", **gint** A tördelés helye az egy sorban megjelenítendő karakterek számával megadva. Az alapértelmezett érték -1, ami jelzi, hogy a szöveget nem kell tördelni.

Azoknak a szöveges cellarajzolóknak a kezelésére, amelyekben a felhasználó megváltoztathatja a szöveget, fontos a következő jelzés.

"edited" Ezt a jelzést a cellarajzó akkor küldi, amikor a felhasználó befejezte a cella tartalmának szerkesztését. Ehhez a jelzéshez csatlakoznunk kell, hogy a raktárban tárolt szöveget megváltoztassuk. A jelzés csatlakoztatható függvény típusa a következő:

```
void név(GtkCellRendererText *cellarajzó, const
gchar *ösvény, const gchar *szöveg, gpointer
adat);
```

Ilyen típusú függvényt kell csatlakoztatnunk az **edited** jelzéshez, hogy a GTK programkönyvtár jelezze a cella megváltoztatása után, hogy mit írt a cellába a felhasználó. A függvény paraméterei a következők:

cellarajzó A szöveg megjelenítő cellarajzó, amelyik az üzenetet létrehozta.

Különösen hasznos ez a paraméter akkor, ha képernyőelem több cellarajzójához is ugyanazt a visszahívott függvényt rendeljük, hiszen a **GtkCellRendererText** típus a **GObject** leszármazottja, így a **g_object_set()** függvénnyel adatokat rendelhetünk hozzá, amit a visszahívott függvényben a **g_object_get()** függvénnyel lekérdezhetünk.

Ha például tudnunk kell a visszahívott függvényben, hogy a raktár melyik oszlopát kell módosítanunk, a cellarajzókhoz hozzárendelhetjük a nekik megfelelő oszlopszámokat.

ösvény Ez a paraméter meghatározza, hogy a cellarajzoló által rajzolt sorok közül melyiket változtatta meg a felhasználó. Ez a szöveges formában megadott ösvény lista- és fa szerkezetű raktárban is használható formában adja meg a szerkesztett cella pontos helyét.

szöveg A felhasználó által beírt szöveg, a cella új tartalma. Ezt a szöveget kell elhelyeznünk a raktárban.

adat A visszahívott függvény nyilvántartásba vételekor megadott mutató, amelyet a programozó olyan érték átadására használ, amilyenre a visszahívott függvényben szüksége van.

A legegyszerűbb programok esetében például arra használhatjuk ezt a mutatót, hogy a raktár memóriabeli címét adjuk.

A szöveges cellarajzoló **"edited"** jelzésének kezelését a 48. példa mutatja be.

A következő példa bemutatja hogyan hozhatunk létre szerkeszthető szöveget tartalmazó cellarajzólót.

48. példa. A következő függvények egy olyan listaszerű képernyőelemet kezelnek, amelyben szerkeszthető szöveges cellarajzoló is van. A függvényekben olyan eszközöket is használunk, amelyeknek részletes leírását csak a későbbiekben adjuk meg.

A példa első függvénye az a visszahívott függvény, amelyet a GTK programkönyvtár akkor hív, amikor a felhasználó befejezte a cella szerkesztését. Ha ezt a függvényt nem hoznánk létre, a felhasználó által létrehozott szöveg nem kerülne vissza a raktárba, így a szerkesztés végén újra megjelenne az eredeti szöveg a képernyőn. Ezt nyilván nem akarjuk, a visszahívott függvényt tehát mindenképpen létre kell hoznunk.

```

1  static void
2  cell_edited(GtkCellRendererText *cell,
3              const gchar          *path_string,
4              const gchar          *new_text,
5              GtkTreeStore         *tree_store)
6  {
7      GtkTreePath *path;
8      GtkTreeIter iter;
9
10     path = gtk_tree_path_new_from_string(path_string);
11     gtk_tree_model_get_iter(GTK_TREE_MODEL(tree_store),
12                             &iter, path);
13     gtk_tree_store_set(tree_store, &iter,

```

278

```

14         0, new_text,
15         -1);
16     gtk_tree_path_free(path);
17 }
```

A függvény első három paramétere – a szöveges cellarajzoló, a szerkesztett cella helyét megadó karakterlánc és a szerkesztés során a felhasználó által beírt új szöveg – az *"edited"* jelzéshez tartozó, a GTK programkönyvtár által adott paraméterek. A harmadik paraméter a fa szerkezetű raktárat jelöli a memóriában. Ezt a paramétert a visszahívott függvény nyilvántartásba vételekor mi magunk adtuk meg.

A függvény a 10. sorban a szerkesztett elem helyét leíró szöveges ösvényleíróból egy ösvényt, a 11–12. sorokban pedig ebből az ösvényből egy bejárót hoz létre.

Ezek után a visszahívott függvény a 13–15. sorokban a bejáró segítségével tárolja a raktárban a felhasználó által beírt új szöveget.

Utolsó lépésként a 16. sorban megsemmisítjük az ösvényt, hiszen arra a továbbiakban már nincsen szükségünk.

A következő függvény egy fa képernyőelem létrehozásakor hívott függvényt mutat be. A függvény elkészíti és beállítja a képernyőelemben megjelenő cellarajzókat. A fa képernyőelemre a későbbiekben, a 298. oldalon található a 8.7. szakaszban visszatérünk.

```

1  void
2  on_editable_text_treeview_realize(
3      GtkWidget *widget,
4      gpointer  user_data)
5  {
6      GtkTreeViewColumn *column;
7      GtkCellRenderer  *renderer;
8
9      /* A raktár. */
10     GtkTreeStore *tree_store;
11     tree_store = gtk_tree_store_new(2,
12         G_TYPE_STRING,
13         G_TYPE_STRING);
14     fill_tree_store(tree_store);
15
16     /* Az első oszlop. */
17     column = gtk_tree_view_column_new();
18     gtk_tree_view_append_column(GTK_TREE_VIEW(widget),
19         column);
20     gtk_tree_view_column_set_title(column, "Cím1");
21 }
```

```

22  /* Az első cellarajzoló. */
23  renderer = gtk_cell_renderer_text_new();
24  g_object_set(G_OBJECT(renderer), "editable",
25             TRUE, NULL);
26  g_signal_connect(renderer, "edited",
27                 G_CALLBACK(cell_edited), tree_store);
28  gtk_tree_view_column_pack_start(column, renderer,
29                                 FALSE);
30  gtk_tree_view_column_add_attribute(column, renderer,
31                                   "text", 0);
32
33  /* A második oszlop. */
34  column = gtk_tree_view_column_new();
35  gtk_tree_view_append_column(GTK_TREE_VIEW(widget),
36                             column);
37  gtk_tree_view_column_set_title(column, "Cím2");
38
39  /* A második cellarajzoló. */
40  renderer = gtk_cell_renderer_text_new();
41  gtk_tree_view_column_pack_start(column, renderer,
42                                 TRUE);
43  gtk_tree_view_column_add_attribute(column, renderer,
44                                   "text", 1);
45
46  gtk_tree_view_set_model(GTK_TREE_VIEW(widget),
47                        GTK_TREE_MODEL(tree_store));
48  }

```

A szerkeszthető cellarajzolót a 23. sorban hozzuk létre, szerkeszthetővé pedig az **"editable"** tulajdonság beállításával a 24–25. sorokban tesszük. A szerkesztésre alkalmas cellarajzoló **"edited"** jelzéséhez a 26–27. sorokban csatlakozunk, ahol megfigyelhetjük a cellarajzoló által megjelenített adatokat hordozó raktár paraméterként történő átadását is.

A függvény további részei ahhoz szükségesek, hogy a két oszlopot megjelenítő cellarajzoló a megfelelő formában megjelenjenek a képernyőn. Ezekre az eszközökre még visszatérünk.

8.4.2. A képmegjelenítő cellarajzoló

A képmegjelenítő cellarajzoló – ahogyan a neve is mutatja – képek megjelenítését teszi lehetővé cellákat tartalmazó képernyőelemeken belül. A GTK programkönyvtár a **GtkCellRendererPixbuf** típust használja a képmegjelenítő cellarajzoló használatára.

280

A képmegjelenítő cellarajzoló létrehozására a következő függvényt használhatjuk:

```
GtkCellRenderer *gtk_cell_renderer_pixbuf_new(void);
```

Ennek a függvénynek a segítségével új képmegjelenítő cellarajzolót hozhatunk létre. A függvény visszatérési értéke az új cellarajzolót jelöli a memóriában.

A következő lista a képmegjelenítő cellarajzoló tulajdonságait, azok neveit és típusát, valamint jelentését mutatja be.

"follow-state", **gboolean** Logikai érték, ami meghatározza, hogy a cella színe – ami függ attól, hogy a cella ki van-e jelölve és az egérkurzor a cella felett található-e – befolyásolja-e a kép színét.

"icon-name", **GCharArray** A megjelenítendő ikon neve.

Ha a cellarajzóval a beállított témának megfelelő ikonokat akarunk megjeleníteni, akkor az adatokat tartalmazó raktárban ikonok nevét kell elhelyeznünk és az oszlopot ehhez a tulajdonsághoz kell kötnünk.

Ez a tulajdonság csak akkor fejt ki hatását, ha a **"stock-id"** és a **"pixbuf"** tulajdonságok nincsenek beállítva.

"pixbuf", **GdkPixbuf** A megjelenítendő kép a GDK programkönyvtár **GdkPixbuf** típusú adatszerkezeteként.

"pixbuf-expander-closed", **GdkPixbuf** A faszerkezetű megjelenítés esetén a leszármazottal is rendelkező sorok előtt egy kép jelenik meg, ami jelzi, hogy az ág becsukott, vagy kinyitott állapotban van-e.

A becsukott állapot jelzésére megjelenő képet ennek a tulajdonságnak a segítségével határozhatjuk meg GDK programkönyvtár adatszerkezetének megadásával.

"pixbuf-expander-open", **GdkPixbuf** A kinyitott állapotot jelző kép (lásd az előző tulajdonságot).

"stock-detail", **GCharArray** FIXME: hogyan is?

"stock-id", **GCharArray** Az előre elkészített és témától függő ikon azonosítóneve.

Ha a képmegjelenítő cellarajzoló segítségével ilyen ikonokat akarunk megjeleníteni, akkor a raktárban az ikonok nevét kell elhelyeznünk és a megfelelő oszlopot ehhez a tulajdonsághoz kell rendelnünk.

"stock-size", **guint** Az előre elkészített ikonok mérete. Ennek a tulajdonságnak a megadásakor a következő állandókat használhatjuk:

GTK_ICON_SIZE_INVALID Ismeretlen ikonméret.

GTK_ICON_SIZE_MENU A menükben, a menüpontok előtt megjelenő ikonoknak megfelelő méret.

GTK_ICON_SIZE_SMALL_TOOLBAR Az eszközsávban megfigyelhető csökkentett méretű ikonoknak megfelelő méret.

GTK_ICON_SIZE_LARGE_TOOLBAR Az eszközsávban megfigyelhető nagyobb méretű ikonoknak megfelelő méret.

GTK_ICON_SIZE_BUTTON A nyomógombokban használt ikonoknak megfelelő méret.

GTK_ICON_SIZE_DND A Drag&Drop műveletek közben az egérmutató mellett megfigyelhető ikonoknak megfelelő méret.

GTK_ICON_SIZE_DIALOG Az üzenetablakokban az üzenet jellegét jelző ikonoknak megfelelő méret.

A képmegjelenítő cellarajzoló használatát a FIXME példa mutatja be részletesebben.

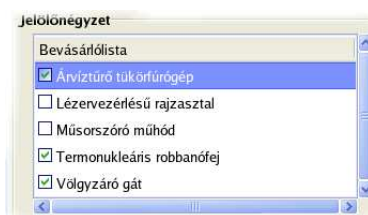
8.4.3. A kapcsoló cellarajzoló

A kapcsoló cellarajzoló logikai igaz, illetve logikai hamis érték megjelenítésére használható. A 8.9. ábrán kapcsoló cellarajzolót láthatunk a sorok elején, a szöveges cellarajzoló előtt.

A kapcsoló cellarajzoló jelölőnégyzetként vagy rádiógombként jeleni meg a képernyőn, jelöli az adott oszlopban található logikai értéket és lehetőséget ad a felhasználónak, hogy az értéket egy egyszerű kattintással megváltoztassa. Magától értetődik, hogy a kapcsoló cellarajzoló akkor lehet különösen hasznos a programozó számára, ha egy időben sok logikai értéket kell megjelenítenie a képernyőn, hiszen a képen is bemutatott módszerrel, egy listában elhelyezve az értékeket, szinte tetszőleges sok érték jeleníthető meg.

A kapcsoló cellarajzoló segítségével ráadásul nem csak jelölőnégyzetként és rádiógombként jeleníthetjük meg a kapcsoló cellarajzolót, használhatjuk ezeknek az képernyőelemeknek az inkonzisztens állapotát is, ami lehetővé teszi a be nem állított állapot jelzését is.

A kapcsoló cellarajzoló kezelésére hasznosak lehetnek a következő függvények:



8.9. ábra. A kapcsoló cellarajzoló

```
GtkCellRenderer *gtk_cell_renderer_toggle_new(void);
```

A függvény segítségével új kapcsoló cellarajzoló készíthetünk. A függvény visszatérési értéke egy mutató, ami az új cellarajzoló jelöli a memóriában.

```
gboolean gtk_cell_renderer_toggle_get_radio(GtkCellRendererToggle *rajzoló);
```

Ennek a függvénynek a segítségével lekérdezhethetjük, hogy a kapcsoló cellarajzoló jelölőnégyzetként vagy rádiógombként fog-e megjelenni a képernyőn.

A függvény argumentuma a cellarajzoló jelöli a memóriában, a visszatérési értéke pedig azt, hogy rádiógombként (logikai *igaz* érték) vagy jelölőnégyzetként (logikai *hamis* érték) jelenik-e meg.

```
void gtk_cell_renderer_toggle_set_radio(GtkCellRendererToggle *rajzoló, gboolean rádiógombként);
```

Ennek a függvénynek a segítségével beállíthatjuk, hogy a kapcsoló cellarajzoló jelölőnégyzetként vagy rádiógombként jelenjen-e meg a képernyőn.

A függvény első paramétere a kapcsoló cellarajzoló jelöli a memóriában, a második paramétere pedig megadja, hogy az rádiógombként (logikai *igaz* érték) vagy jelölőnégyzetként (logikai *hamis* érték) jelenjen-e meg a képernyőn. A kapcsoló cellarajzók alapértelmezés szerint jelölőnégyzetként jelennek meg a képernyőn, ha ez megfelel a céljainknak, nem kell ezt a függvényt használnunk.

Tudnunk kell azt is, hogy a kapcsoló jelölőnégyzet megjelenése nem módosítja a viselkedését, sőt maga a cellarajzoló nem kapcsolódik át amikor a felhasználó rákattint, egyszerűen csak meghívja a megfelelő visszahívott függvényt. A programozónak kell gondoskodnia arról, hogy a visszahívott függvényben megváltoztassa a kapcsoló cellarajzoló értékét, ahogyan az is, hogy az átkapcsoló rádiógomb-szerűen kikapcsolja az esetleg bekapcsolt egyéb rádiógombokat.

A kapcsoló cellarajzók esetében a következő tulajdonságokat érdemes ismernünk:

"activatable", **gboolean** Logikai érték, ami megadja, hogy az adott kapcsoló cellarajzoló a felhasználó átkapcsolhatja-e, azaz az adott kapcsológomb érzékeny legyen-e.

Az alapértelmezett érték a **TRUE**, ami azt jelzi, hogy a kapcsológombra kattintáskor meghívódik a megfelelő visszahívott függvény, amiben a programozónak magának kell gondoskodnia az átkapcsolásról.

"active", **gboolean** Logikai érték, ami megadja a kapcsoló cellarajzoló állapotát. Amikor lista vagy fa szerkezetű raktár logikai értéket tároló oszlopainak megjelenítésére használjuk a cellarajzólót, ezt a tulajdonságot hozzá szoktuk rendelni a raktár valamelyik oszlopához.

E tulajdonság alapértelmezett értéke a **FALSE**, amit nyilván ritkán használunk, hiszen az állapotot mindig magunk állítjuk be.

"inconsistent", **gboolean** Logikai érték, ami megadja, hogy a cellarajzoló inkonzisztens állapotban van-e. Ha a kapcsoló cellarajzoló inkonzisztens állapotát is használja a programunk, akkor szerencsés ezt a tulajdonságot is hozzá rendelni a raktár egy oszlopához.

E tulajdonság alapértelmezett értéke a **FALSE**, ami jelzi, hogy a cellarajzoló nincs inkonzisztens állapotban és ami így általában meg is felel az igényeinknek.

"radio", **gboolean** Logikai érték, ami jelzi, hogy a kapcsoló cellarajzoló rádiógombként jelenik-e meg, vagy jelölőnégyzetként.

Az alapértelmezett érték a **FALSE**, ami jelzi, hogy a kapcsoló cellarajzoló jelölőnégyzetként jelenik meg a képernyőn.

A kapcsoló cellarajzoló használatához ismernünk kell a következő jelzést:

toggled Ezt a jelzést akkor küldi a kapcsoló cellarajzoló, amikor a felhasználó kattintással átkapcsolja az állapotát. Mivel a cellarajzoló „magától” nem változtatja meg az állapotát, ezt a jelzést mindenképpen kezelniünk kell, ha a cellarajzólót átkapcsolható módon akarjuk kezelni a programunkban.

A jelzéshez használható visszahívott függvény típusa a következő:

```
void név(GtkCellRendererToggle *cellarajzoló, gchar
*ösvény, gpointer adat);
```

A függvény paramétere a következők:

cellarajzoló Ez a paraméter egy mutató, ami azt a cellarajzólót jelöli a memóriában, amely a jelzést létrehozta. Ha a visszahívott függvény több cellarajzoló üzeneteit is kezeli (azaz több oszlop jelzéseit is fogadja), arra használhatjuk ezt a paramétert, hogy azonosítsuk az oszlopot például úgy, hogy a cellarajzolóhoz a létrehozásakor adatokat rendelünk a **g_object_set_data()** függvénnyel.

ösvény Ez a karakterlánc megadja a konkrét cellához vezető ösvényt, ami lista és fa szerkezetű raktárakban is képes egyértelműen azonosítani, hogy a felhasználó melyik sorra kattintott.

adat Ez a paraméter egy mutató, azt az értéket adja meg, amit a visszahívott függvény nyilvántartásba vételekor megadtunk.

A legtöbb esetben arra használjuk ezt a paramétert, hogy a visszahívott függvénynek átadjuk a raktár címét, ami az adatokat tartalmazza.

A következő példa változtatható értékű kapcsoló cellarajzolókat mutat be. A példa sorai tartalmazznak olyan függvényhívásokat is, amelyeket csak későbbiekben mutatunk be részletesebben, de a magyarázat alapján a működése ennek ellenére megérthető.

49. példa. A következő függvény egy lista képernyőelemet valósít meg. A lista első oszlopa a átkapcsolható jelölőnégyzeteket, második oszlopa szöveges magyarázatot tartalmaz. A program képernyőképe a 8.9. ábrán látható.

```

1  static void
2  cell_toggled(GtkCellRendererToggle *cell,
3               gchar                  *path_str,
4               GtkTreeModel           *model)
5  {
6      GtkTreeIter iter;
7      gboolean value;
8      GtkTreePath *path =
9          gtk_tree_path_new_from_string(path_str);
10
11     gtk_tree_model_get_iter(model, &iter, path);
12     gtk_tree_model_get(model, &iter,
13                         0, &value, -1);
14
15     value ^= 1;
16
17     gtk_list_store_set(GTK_LIST_STORE (model), &iter,
18                       0, value, -1);
19     gtk_tree_path_free(path);
20 }
21
22 void
23 on_toggle_treeview_realize(GtkWidget *widget,
24                             gpointer   user_data)
25 {
26     GtkTreeViewColumn *column;
27     GtkCellRenderer   *renderer;
28     GtkTreeIter iter_level0;

```

```

29
30     /* A raktár. */
31     GtkTreeStore *list_store;
32     list_store = gtk_list_store_new(2,
33                                     G_TYPE_BOOLEAN,
34                                     G_TYPE_STRING);
35     fill_list_store(list_store);
36
37     /* Az oszlop. */
38     column = gtk_tree_view_column_new();
39     gtk_tree_view_append_column(GTK_TREE_VIEW(widget),
40                                 column);
41     gtk_tree_view_column_set_title(column,
42                                     "Bevásárlólista");
43
44     /* Az első cellarajzoló. */
45     renderer = gtk_cell_renderer_toggle_new();
46     g_signal_connect(renderer, "toggled",
47                     G_CALLBACK(cell_toggled), list_store);
48     gtk_tree_view_column_pack_start(column, renderer,
49                                     FALSE);
50     gtk_tree_view_column_add_attribute(column, renderer,
51                                     "active", 0);
52
53     /* A második cellarajzoló. */
54     renderer = gtk_cell_renderer_text_new();
55     gtk_tree_view_column_pack_start(column,
56                                     renderer, TRUE);
57     gtk_tree_view_column_add_attribute(column, renderer,
58                                     "text", 1);
59
60     gtk_tree_view_set_model(GTK_TREE_VIEW(widget),
61                             GTK_TREE_MODEL(list_store));
62 }

```

A példa 22–62. sorai közt figyelhetjük meg a fa képernyőelem létrehozásakor visszahívott függvényt, ami a képernyőelem megjelenítése előtt elkészíti és beállítja a képernyőn megjelenítendő oszlopokat.

A függvény a 32–34. sorokban létrehoz egy lista típusú képernyőelemet, amelynek első oszlopa logikai, második oszlopa pedig karakterlánc típusú értéket hordoz.

A függvény 38–42. soraiban létrehozunk egy oszlopot, amelyet aztán hozzárendelünk a fa képernyőelemhez. Ezekre a lépésekre a későbbiekben, a fa képernyőelem bemutatása során részletesen is visszatérünk.

A kapcsoló cellarajzoló a 45. sorban hozzuk létre. A 46–47. sorok közt a kapcsoló cellarajzó **toggled** jelzéshez hozzárendeljük a **cell_toggled()** függvény hívását. Figyeljük meg, hogy a nyilvántartásba vett visszahívott függvénynek paraméterként minden híváskor átadjuk a **list_store** mutató értékét, így a visszahívott függvény képes lesz megváltoztatni a lista szerkezetű raktár tartalmát.

A függvény a 48. sorban a létrehozott cellarajzolóhoz hozzárendeli a képernyőelem oszlopához. Erre a lépésre a fa képernyőelem bemutatása során visszatérünk, ahogyan a függvény 50–51. sorában megfigyelhető függvényhívásra is, amelynek során a kapcsoló cellarajzó állapotát (a **"active"** tulajdonság) hozzárendeljük a lista szerkezetű raktár első oszlopához.

A függvény további soraiban először elkészítjük és beállítjuk a szöveges cellarajzoló (53–58. sorok), majd a lista típusú raktárat hozzárendeljük a fa képernyőelemhez.

A példa 1–20. soraiban található a visszahívott függvény, amit a 46–47. sorokban vettünk nyilvántartásba, hogy a GTK programkönyvtár meghívja, amikor a kapcsoló cellarajzoló a felhasználó átkapcsolja.

A függvény a 8–9. sorokban a paraméterként kapott szöveges leíróból létrehoz egy ösvényt ami a felhasználó által aktivált cellát azonosítja. A függvény ezek után a 11. sorban beállítja a helyi változóként létrehozott fabejárót a raktár kezelésére.

Ezek után a következő lépés a raktár olvasása (12–13. oldal), a logikai érték ellentettjének kiszámítása (15. sor) és az érték visszaírása (17–18. sor). Figyeljük meg, hogy a függvény a raktár legelső (0 sorszámú) oszlopát olvassa és írja, ami jelzi, hogy csak egy cellarajzó, a legelső oszlopot kezelő cellarajzó kezelésére van felkészítve.

Ezek után a függvény a 19. sorban megsemmisíti a már szükségtelen ösvényt.

A visszahívott függvény kapcsán megfigyelhetjük, hogy az kizárólag a raktár adott elemének ellentettjére állítását végzi el, nem küld üzenetet a módosításról, nem gondoskodik arról, hogy a képernyő kövesse a változtatást. Erre nincs is szükség, a cellarajzó automatikusan követi a raktár módosításait.

8.4.4. A folyamatjelző cellarajzó

A folyamatjelző cellarajzó segítségével a felhasználót több egyidőben zajló esemény előreheledtáról tájékoztathatjuk. Ilyen jellegű eszközre viszonylag ritkán van szükségünk, ha azonban sok szálon futó programot készítünk ez az eszköz nagymértékben növelheti programunk használhatóságát. A folyamatjelző cellarajzó létrehozására a következő függvényt használhatjuk:

`GtkCellRenderer *gtk_cell_renderer_progress_new(void);` A függvény segítségével új folyamatjelző cellarajzolóhoz hozhatunk létre. A függvény visszatérési értéke az új cellarajzoló jelölő a memóriában.

A folyamatjelző cellarajzoló legfontosabb tulajdonságait és azok típusát a következő lista mutatja be.

"text", gchararray A folyamatjelző cellarajzoló területén megjelenő szöveg. Ha ezt a tulajdonságot beállítjuk a cellarajzoló által elfoglalt területet kétszeresen is hasznosíthatjuk.

"value", gint A folyamatjelző cellarajzolóban megjelenített szám jellegű érték, ami 0 és 100 között a cellában megjelenő sáv hosszát adja meg százalékos formában.

8.5. A cellaelrendezés

A cellarajzolókat – ahhoz, hogy megjelenjenek – el kell helyeznünk a képernyőn, erre pedig a cellaelrendezéseket használhatjuk. A GTK programkönyvtár a cellaelrendezés kezelésére a `GtkCellLayout` típust hozza létre. A `GtkCellLayout` egy interfész, amelyet mindazoknak az eszközöknek meg kell valósítaniuk, amelyek cellarajzoló segítségével kívánnak adatokat megjeleníteni.

A cellarajzoló fontos, de nem az egyetlen tulajdonsága az, hogy több cellarajzoló befogadására is alkalmas, így egyetlen cellaelrendezésen belül több cellarajzoló is megjeleníthető. A cellaelrendezés másik fontos képessége az, hogy a benne elhelyezett cellarajzoló tulajdonságait – amelyeket a konkrét cellarajzoló esetében már részletesen bemutatunk – képes raktárakból, azok megfelelő sorainak és oszlopainak olvasásával beállítani. Ennek a képességnek köszönhető, hogy a cellaelrendezésben a cellarajzoló az egyes sorokban vagy oszlopokban különféle adatokat képesek megjeleníteni.

A cellarajzoló használatát a következőképpen foglalhatjuk össze. Először létre kell hoznunk egy raktárat, amelyben elhelyezzük a megjelenítendő adatokat, majd létre kell hoznunk a cellarajzolókat, amelyek megjelenítik ezeket az adatokat. A következő lépésként olyan képernyőelemre van szükségünk, amely megvalósítja a cellaelrendezés interfészt, majd ebben a cellaelrendezésben el kell helyeznünk a cellarajzolókat. Utolsó lépésként a cellarajzoló megfelelő függvényeinek segítségével be kell állítanunk, hogy az a cellarajzoló mely tulajdonságait olvassa a raktár egyes oszlopaiból. A munka kétségtelenül bonyolult egy kissé, viszont igen rugalmasan használható eszközként összetett, az igényeinkhez alakítható felhasználói felületet eredményez.

A következő függvények segítségével a cellaelrendezésben a cellarajzók elhelyezhetők és a raktár felé a megfelelő kapcsolatok felépíthetők:

```
void gtk_cell_layout_pack_start(GtkCellLayout
    *cellaelrendezés, GtkCellRenderer *cellarajzó,
    gboolean nyújt);
```

A függvény segítségével a cellaelrendezésben új cellarajzót helyezhetünk el. Egy cellaelrendezésben több cellarajzó is lehet, a cellarajzó azonban csak egy cellaelrendezésben jeleníthető meg. E függvény a cellarajzót a cellaelrendezés elején, annak első elemeként szúrja be.

A függvény első paramétere a cellaelrendezést, második paramétere pedig a cellarajzót jelöli a memóriában.

A függvény harmadik paramétere logikai érték, ami megadja, hogy az adott cellarajzó méretét szükség esetén nyújtani lehet-e. Ha ez a paraméter **TRUE** értékű, a cellarajzó mérete növelhető, ha **FALSE**, akkor nem. A cellaelrendezés az esetleges helytöbbletet egyenlő arányban osztja el a nyújtható módon elhelyezett cellarajzók közt.

```
void gtk_cell_layout_pack_end(GtkCellLayout
    *cellaelrendezés, GtkCellRenderer *cellarajzó,
    gboolean nyújt);
```

E függvény működése megegyezik a `gtk_cell_layout_pack_start()` függvény működésével, de a cellarajzót nem a cellaelrendezés elejére, hanem a végére helyezi el.

```
void gtk_cell_layout_set_attributes(GtkCellLayout
    *cellaelrendezés, GtkCellRenderer *cellarajzó, ...,
    NULL);
```

A függvény segítségével beállíthatjuk, hogy az egyes benne elhelyezett cellarajzóknak milyen módon jelenítse meg a raktár elemeit. A függvény a cellaelrendezés számára esetlegesen már beállított összes kapcsolatokat megszünteti és a paraméterként megadott értékekkel helyettesíti.

A függvény első paramétere a cellaelrendezést, a második paramétere pedig a cellarajzót jelöli a memóriában.

A függvény további paramétere párok, amelyeknek első tagja az adott cellarajzó tulajdonságának neve, második tagja pedig egy szám, ami megadja, hogy az adott tulajdonságot a használt raktár melyik oszlopa alapján kell változtatni.

A függvény paramétereiként megadott párok végén egy **NULL** értéket kell megadnunk, ellenkező esetben a program futása megszakadhat.

```
void gtk_cell_layout_add_attribute(GtkCellLayout
    *cellaelrendezés, GtkCellRenderer *cellarajzó, const
```

`gchar *tulajdonségnév, gint oszlopszám);` A függvény működése hasonlít a `gtk_cell_layout_set_attributes()` függvény működésével, de e függvény segítségével a kapcsolatokat egyenként, fokozatosan vehetjük nyilvántartásba, mert az a függvény a meglévő kapcsolatokat nem törli, csak egy új kapcsolatot vesz nyilvántartásba. Nyilvánvaló, hogy ezt a függvényt akkor célszerű használni, ha a cellarajzoló állapotát csak egy érték módosítja.

A cellaelrendezés használatát a következő példa mutatja be, ahol cellaelrendezésként egy, a későbbiekben bemutatott típust használunk.

50. példa. A következő programrészlet a cellaelrendezés használatát mutatja be. Az alábbi sorok egy cellaelrendezést hoznak létre és abban két cellarajzolt helyeznek el, valamint megteremtik a kapcsolatot a raktár és a cellaelrendezés közt.

```

1  col = gtk_tree_view_column_new();
2  gtk_tree_view_append_column(GTK_TREE_VIEW(widget), col);
3  gtk_tree_view_column_set_title(col, "Cím");
4
5  renderer = gtk_cell_renderer_pixbuf_new();
6  gtk_tree_view_column_pack_start(col, renderer, FALSE);
7  gtk_tree_view_column_add_attribute(col, renderer,
8    "pixbuf", 0);
9
10 renderer = gtk_cell_renderer_text_new();
11 gtk_tree_view_column_pack_start(col, renderer, TRUE);
12 gtk_tree_view_column_add_attribute(col, renderer,
13   "text", 1);

```

A programrészlet 1. sorában egy, a képernyőn megjeleníthető fa képernyőelem oszlopát reprezentáló adatszerkezetet hozunk létre. Ezt az eszközt a későbbiekben a 8.7. szakaszban mutatjuk be, egyelőre elég annyit tudnunk róla, hogy cellaelrendezésként is használható.

A részlet 2. sorában a cellarajzolóként használható oszlopot elhelyezzük egy képernyőlemben, hogy a képernyőn megjelenjen, majd a 3. sorban beállítjuk a címét. Ezek a lépések egy konkrét programban fontosak, a cellaelrendezések működésének megértése szempontjából azonban lényegtelenek.

Előfordulhat, hogy az adatokat tartalmazó raktár egyes oszlopai és a cellarajzoló tulajdonságai közt nem tudunk egyszerű kapcsolatot megfogalmazni, azaz a megjelenítés mikéntjét nem tudjuk a `gtk_tree_view_column_add_attribute()` `gtk_cell_layout_set_attributes()` függvény segítségével megfogalmazni.



Megnevezés	Költség	Mb
Önindító javítása		☐
Munkabér	2000 Ft	☒
Alkatrész költség	5500 Ft	☐
<i>Kattintson ide új kiadás beírásához.</i>		
Teljes költség:	7500 Ft	☐

8.10. ábra. A cella adatfüggvény hatása

Néha a megjelenítendő adatokat a megjelenítés előtt át kell alakítanunk. Egy konkrét alkalmazás esetében például könnyen lehet, hogy a raktárban tárolt számokat mértékegységgel akarjuk megjeleníteni egy saját függvény meghívásával. Az is lehet, hogy az adatokat az értéküktől függően más-más formában akarjuk a képernyőre rajzolni, lehetséges például, hogy a negatív számokat piros, míg a pozitív számokat fekete betűkkel akarjuk szedni, hogy bizonyos sorokra felhívjuk a figyelmet.

Ezekre az esetekre láthatunk példát a 8.10. ábrán. Az ábra költség oszlopa szám jellegű értéket tartalmaz, a képernyőn való megjelenítéskor azonban a pénznemet is utána kell írunk. A 0 értéket tartalmazó soroknál viszont mind a pénznem, mind pedig a szám elmarad, a szöveges cellarajzolóval egyetlen betűt sem rajzolunk. Ráadásul az oszlop betűtípusa is változik annak függvényében, hogy milyen jellegű sorban vagyunk. Az összesített értéket jelző sorban félkövér dőlt betűvel szedjük az értéket, ami nyilvánvalóan sokkal olvashatóbbá teszi a képernyőn megjelenített adatokat.

Ilyen jellegű megoldásokat úgy készíthetünk, hogy a megjelenített adatok és a cellarajzoló közé „beékelünk” egy saját készítésű függvényt, ami minden sorra kiolvassa a raktár adatait és beállítja a cellarajzoló tulajdonságait. A függvényt el kell készítenünk, majd a cellaelrendezés létrehozása után nyilvántartásba kell vetetnünk. Ehhez a következő eszközökre van szükségünk.

```
void gtk_cell_layout_set_cell_data_func(GtkCellLayout
    *cellaelrendezés, GtkCellRenderer *cellarajzoló,
    GtkCellLayoutDataFunc függvény, gpointer adatok,
    GDestroyNotify felszabadító);
```

Ennek a függvénynek a segítségével a raktár adatai és a cellarajzoló közés saját készítésű cella adatfüggvényt helyezhetünk. A cella adatfüggvényt a GTK minden megjelenítendő sorra meghívja, feladata, hogy a raktárból az adatokat kiolvassa és a cellarajzoló tulajdonságait – melyek meghatározzák a megjelenítendő adatokat és a megjelenítés formáját is – beállítsa.

A függvény első paramétere a cellaelrendezést, második paramétere pedig a cellarajzólót jelöli a memóriában. A függvény harmadik paramétere a cella adatfüggvény címe, negyedik paramétere pedig a cella adatfüggvénynek átadandó kiegészítő adatokat tartalmazó memóriaterületet jelöli a memóriában. Ez utóbbi paraméter értéke lehet `NULL`, ha a szokásos paramétereken kívül nem akarunk mást

átadni a cella adatfüggvénynek.

A függvény ötödik paramétere egy olyan függvény, ami a cella adatfüggvénynek átadandó kiegészítő adatokat felszabadító függvény, amelyet a GTK akkor hív, ha a kiegészítő adatokra már nincs szükség. Ha ilyen felszabadító függvényt nem akarunk használni ez a paraméter is lehet `NULL`.

```
void függvénynév(GtkCellLayout *cellaelrendezés,
GtkCellRenderer *cellarajzoló, GtkTreeModel *raktár,
GtkTreeIter *bejáró, gpointer adatok);
```

Ilyen típusú függvényt kell készítenünk cella adatfüggvényként.

A függvény első és második paramétere a cellaelrendezést és a cellarajzolót jelöli a memóriában. A harmadik paraméter a raktár címe, ahonnan az adatokat a függvénynek ki kell olvasnia.

A negyedik paraméter egy bejárót jelöl a memóriában. A bejáró a raktár azon sorát azonosítja, amelynek rajzolására éppen szükség van. Azt, hogy a raktár melyik oszlopaiból olvassa a cella adatfüggvény az adatokat, nekünk kell eldöntenünk. A legtöbb esetben több oszlop értékére is szükségünk van annak eldöntésére, hogy mi – és főleg milyen formában – jelenjen meg a cellarajzolóban.

A függvény ötödik paramétere a kiegészítő adatokat jelöli a memóriában. A kiegészítő adatokat a cella adatfüggvény nyilvántartásba vételekor határozhatjuk meg.

```
void függvény(gpointer adatok);
```

Ilyen típusú függvényt kell felszabadító függvényként készíteni, ha a kiegészítő adatok automatikus felszabadítására is szükségünk van.

A függvény paramétere a felszabadítandó kiegészítő adatokat jelöli a memóriában. Láthatjuk, hogy felszabadító függvényként egyszerű esetben akár a `g_free()` függvény is megadható.

A következő példa a cella adatfüggvény elkészítését és megvalósítását mutatja be a 8.10. ábrán. látható képernyőképet előállító program egy részletének segítségével.

51. példa. A következő sorok egy cella adatfüggvényt mutatnak be. A függvény egyszerű, kiegészítő adatokat nem fogad, a szöveges cellarajzoló által megjelenítendő szöveget és annak formáját határozza meg.

```
1 static void
2 render_cost(
3     GtkCellLayout *cell_layout,
4     GtkCellRenderer *cell,
```


292

```

5      GtkTreeModel      *tree_model,
6      GtkTreeIter       *iter,
7      gpointer          data)
8  {
9      gboolean placeholder;
10     gboolean commission;
11     gboolean sumline;
12     gchar    *designation;
13     gint     cost;
14     gchar    *text;
15
16     gtk_tree_model_get(tree_model, iter,
17         ColumnIsPlaceholder, &placeholder,
18         ColumnIsCommission, &commission,
19         ColumnDesignation, &designation,
20         ColumnCost,        &cost,
21         ColumnIsTotal,     &sumline,
22         -1);
23
24     text = g_strdup_printf(_("%d Ft"), cost);
25
26     g_object_set(G_OBJECT(cell),
27         "text", cost <= 0 ? "" : text,
28         "foreground", placeholder ? "Gray" : "Black",
29         "editable", !commission && !sumline &&
30             !placeholder,
31         "weight", sumline ? PANGO_WEIGHT_BOLD :
32             PANGO_WEIGHT_NORMAL,
33         "style", sumline ? PANGO_STYLE_ITALIC :
34             PANGO_STYLE_NORMAL,
35         NULL);
36
37     g_free(text);
38 }

```

A függvény 16–22. soraiban a raktár éppen rajzolt sorának több oszlopát is olvassuk, hogy eldöntsük mit és hogyan kell megjeleníteni a cellarajzolóban. A 26–35. sorokban a cellarajzoló tulajdonságait állítjuk be egy lépésben. A tulajdonságok közt a 27. sorban megtalálható a megjelenítendő szöveg is, ami az üres karakterlánc, ha a megjelenítendő szám 0 vagy kisebb értékű.

A cella adatfüggvény nyilvántartásba vételét mutatja be következő néhány sor.

```

1  renderer = gtk_cell_renderer_text_new();
2  gtk_tree_view_column_pack_start(column, renderer,
3      FALSE);
4  gtk_cell_layout_set_cell_data_func(
5      GTK_CELL_LAYOUT(column),
6      renderer,
7      (GtkCellLayoutDataFunc ) render_cost,
8      NULL, NULL);

```

Az 1. sorban létrehozuk a cellarajzoló, a 2–3. sorban nyilvántartásba vesszük, a 4–8. sorokban pedig bejegyezzük a cella adatfüggvényt. A 8. sorban megfigyelhetjük, hogy sem kiegészítő adatokra, sem pedig az azokat felszabadító függvényre nincs szükségünk.

A cella adatfüggvény kapcsán érdemes megemlítenünk, hogy azt a GTK a cella rajzolása előtt mindig hívja, érdemes tehát egyszerű, gyors függvényt használni. Nyilvánvaló, hogy érdemes úgy megtervezni a raktárban tárolt adatszerkezeteket, hogy a cella adatfüggvény egyszerűen elvégezhesse a feladatát, azaz amit csak lehet érdemes előre kiszámítani.

8.6. A kombinált doboz

A kombinált doboz használatának alapjait már bemutattuk a 3.1.7. oldalon, ahol az egyszerűség kedvéért kizárólag az egyszerű, csak szöveges adatokat megjelenítő változatokkal foglalkoztunk. A kombinált dobozban azonban különféle cellarajzolókat is elhelyezhetünk, így az egyszerű lista helyett sokkal kifejezőbb – és tegyük hozzá szebb – képernyőelemet nyerhetünk. Ezt mutatja be a 8.11. ábra, ahol képet és szöveget egyaránt megjelenítő kombinált dobozt láthatunk.

A kombinált doboz kezelésére a GTK programkönyvtár – ahogyan azt már említettük – a `GtkComboBox` típust használja. A Glade program által létrehozott programok azonban az ilyen képernyőelemeket a `gtk_combo_box_new_text()` függvénnyel hozza létre. Az e függvénnyel létrehozott kombinált dobozokban azonban csak szöveges értékek jeleníthetők meg. Ha összetettebb megjelenésű kombinált dobozokat akarunk használni, akkor azokat magunknak kell elkészítenünk.

Szerencsére a Glade lehetőséget biztosít arra, hogy az általa nem támogatott képernyőelemeket is használjuk a felhasználói felületbe. Ha ilyen terveink vannak, akkor a képernyőelemek szerkesztése közben *egyedi* (*custom-made*, rendelésre készített) képernyőelemet kell használnunk.



8.11. ábra. A kombinált doboz

Ehhez az ablakban a *egyéni felületi elem* néven szereplő képernyőelemet kell elhelyeznünk, és meg kell írunk a *létrehozó függvényt*, ami a képernyőelemet létrehozza és visszaadja az új képernyőelem memóriacímét. Igen fontos, hogy a létrehozó függvény egy `GtkWidget` típusú elemet jelölő memóriacímét adjon vissza, mert ha nem a megfelelő memóriacímét kapja a Glade által készített, az adott ablakot létrehozó függvény, a program futása megszakad.

A kombinált doboz képernyőelemet ábrázoló `GtkComboBox` megvalósítja a `GtkCellLayout` interfészt, ezért a benne megjelenő adatokat cellarajzolók segítségével ábrázolhatjuk. Ehhez a már bemutatott eszközökön kívül a következő függvényekre van szükségünk.

`GtkWidget* gtk_combo_box_new(void);` E függvénnyel kombinált dobozt hozhatunk létre. Azért kell ezt a függvényt használnunk, mert a Glade a `gtk_combo_box_new_text()` függvénnyel hozza létre a kombinált dobozokat, így azokat csak szöveges értékek megjelenítésére használhatjuk.

A függvény visszatérési értéke az új kombinált dobozt jelöli a memóriában.

`void gtk_combo_box_set_model(GtkComboBox *doboz, GtkTreeModel *raktár);` A függvény segítségével beállíthatjuk, hogy a kombinált dobozban melyik raktár sorai jelenjenek meg.

A függvény első paramétere a kombinált dobozt, második paramétere pedig a raktárat jelöli a memóriában.

`gboolean gtk_combo_box_get_active_iter(GtkComboBox *doboz, GtkTreeIter *bejáró);` A függvény segítségével lekérdezhetjük, hogy a felhasználó a raktár melyik elemét választotta ki a kombinált doboz segítségével.

A függvény első paramétere a kombinált dobozt jelöli a memóriában. A második paraméter arra a bejáróra mutat, amelyet az épen kiválasztott elemre akarunk állítani. A függvény visszatérési értéke igaz logikai értéket képvisel, ha a kombinált dobozban volt kiválasztott elem, így a bejárót sikerült beállítani. Ha nem volt kiválasztott elem, a visszatérési érték hamis.

A következő példaprogram a ?? ábrán látható összetett megjelenésű kombinált doboz létrehozását mutatja be.

52. példa. *E példa bemutatja hogyan készíthetünk olyan kombinált dobozt, amiben a választható elemeket egy kép és a mellette megjelenő szöveg jelképezi.*

Az első függvény egy lista szerkezetű raktárat készít, amiben két oszlop található. Az első oszlop egy képet, a második pedig egy szöveges értéket tartalmaz.

```

1  typedef struct _country country;
2  struct _country {
3      gchar *short_code;
4      gchar *name;
5  };
6
7  static GtkListStore *
8  languages_list_store_new(void)
9  {
10     GtkListStore *list_store;
11     GdkPixbuf *pixbuf;
12     GtkTreeIter tree_iter;
13     gint n;
14     gchar *file;
15     country countries[] = {
16         { "HU", "Magyar" },
17         { "DE", "Német" },
18         { "UK", "Angol" },
19         { "US", "Amerikai Angol" },
20         { NULL, NULL }
21     };
22     /*
23      * A raktár létrehozása.
24      */
25     list_store = gtk_list_store_new(2,
26         G_TYPE_OBJECT,
27         G_TYPE_STRING);
28     /*
29      * A raktár feltöltése.
30      */
31     for (n = 0; countries[n].short_code != NULL; ++n) {
32         file = g_strdup_printf(PIXMAP_DIR "flag-%s.png",
33             countries[n].short_code);
34         pixbuf = gdk_pixbuf_new_from_file(file, NULL);
35         gtk_list_store_append(list_store, &tree_iter);
36         gtk_list_store_set(list_store, &tree_iter,
37             0, pixbuf,
38             1, countries[n].name,
39             -1);
40         g_free(file);

```

296

```

41     }
42
43     return list_store;
44 }
```

A függvény a 15–20. sorok közt olvasható értékeket helyezi el a 25–27. sorokban létrehozott raktárban, azaz igazából a 34. sorban létrehozott képet, amelyet az eredeti adatszerkezetben elhelyezett név alapján hoz létre. Ezt a függvényt receptként használhatjuk állandó adatokat tartalmazó lista szerkezetű raktárak létrehozására.

A következő függvényt a kombinált doboz létrehozására használjuk. A függvény első paramétere a létrehozandó képernyőelem nevét határozza meg, míg a többi paraméterét a Glade programban határozhatjuk meg. A függvény visszatérési értéke igen fontos, a függvény által létrehozott képernyőelemet jelöli a memóriában. A Glade ilyen függvényeket használ a programozó által létrehozott, egyedi képernyőelemek előállítására.

```

1  GtkWidget*
2  create_combo_box_complex(
3      gchar *widget_name,
4      gchar *string1,
5      gchar *string2,
6      gint int1,
7      gint int2)
8  {
9      GtkWidget      *widget;
10     GtkCellRenderer *renderer;
11     GtkListStore    *list_store;
12     /*
13      * A raktár és a kombinált doboz létrehozása.
14      */
15     list_store = languages_list_store_new();
16     widget = gtk_combo_box_new();
17     /*
18      * Az első cellarajzoló.
19      */
20     renderer = gtk_cell_renderer_pixbuf_new();
21     gtk_cell_layout_pack_start(GTK_CELL_LAYOUT(widget),
22                               renderer, FALSE);
23     gtk_cell_layout_add_attribute(GTK_CELL_LAYOUT(widget),
24                                   renderer, "pixbuf", 0);
25     /*
26      * A második cellarajzoló.
27      */
```

```

28     renderer = gtk_cell_renderer_text_new();
29     gtk_cell_layout_pack_start(GTK_CELL_LAYOUT(widget),
30         renderer, TRUE);
31     gtk_cell_layout_add_attribute(GTK_CELL_LAYOUT(widget),
32         renderer, "text", 1);
33     /*
34      * A raktár beállítása.
35      */
36     gtk_combo_box_set_model(GTK_COMBO_BOX(widget),
37         GTK_TREE_MODEL(list_store));
38     gtk_combo_box_set_active(GTK_COMBO_BOX(widget), 0);
39
40     return widget;
41 }

```

A függvényben előbb létrehozuk a raktárat és a kombinált dobozt a 15., illetve a 16. sorokban. A függvény további soraiban két cellarajzolóhozunk létre, egyet kép, egyet pedig szöveg megjelenítésére a 20., illetve a 28. sorban.

A cellarajzolókat ezeket után a `gtk_cell_layout_pack_start()` függvény segítségével elhelyezzük az előzőleg létrehozott kombinált dobozban (21–22. és 29–30. sorok). Figyeljük meg, hogy mivel a kombinált doboz megvalósítja a `GtkCellLayout` interfészt, a `GTK_CELL_LAYOUT()` típuskényszerítő makrót használhatjuk a függvény hívásakor! Megfigyelhetjük azt is, hogy a szöveget nyújtható módon, a képet viszont állandó szélességgel helyezzük el a cellaelrendezésben. Ez célszerű döntés, hiszen így a szöveg balra zártan, egy vonalban kezdődve jelenik meg minden sorban.

A következő feladat a megjelenítendő oszlopok kijelölése, amelyet a `gtk_cell_layout_add_attribute()` függvénnyel végzünk el a két cellarajzolóval a 23–24., illetve a 31–32. sorokban. Figyeljük meg, hogy a beállítások alapján a cellaelrendezés a kép jellegű cellarajzoló `"pixbuf"` tulajdonságát a 0, a szöveges cellarajzoló `"text"` tulajdonságát pedig az 1 sorszámú oszlopból veszi.

Igen fontos a 36–37. sorokban olvasható függvényhívás, amelynek a segítségével megadjuk, hogy a kombinált dobozban melyik raktár adatai jelenjenek meg.

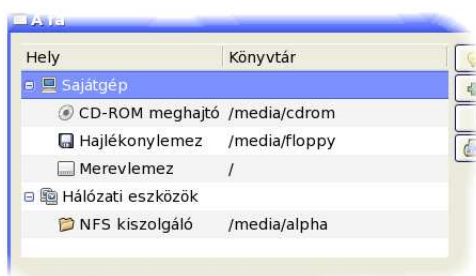
Fontosak viszont az ezek után látható függvényhívások. Az 5. sorban egy kép jellegű cellarajzólót, a 10. sorban pedig egy szöveges cellarajzólót hozunk létre. Mindkét cellarajzólót elhelyezzük a cellaelrendezésben. A képet megjelenítő cellarajzólót a 6. sorban nem nyújtható módon, a szöveges cellarajzólót pedig a 11. sorban nyújtható módon helyezzük el a cellaelrendezésben.

A következő fontos lépés cellarajzoló és az általa megjelenítendő adatok közti kapcsolat felépítése. A kép jellegű cellarajzoló `"pixbuf"` nevű

tulajdonságát a megjelenítendő raktár 0 sorszámu oszlopához kötjük a 7–8. sorokban, míg a szöveges cellarajzoló **"text"** nevű tulajdonságát a raktár 1 sorszámu oszlopából vesszük a 12–13. sor szerint.

8.7. A fa képernyőelem

A fa képernyőelem a GTK programkönyvtár összetettebb képernyőelemeinek egyike, listák, fák, szöveges, illetve grafikus adatok megjelenítésére egyaránt alkalmas. A fa képernyőelem segítségével a programunkat könnyen kezelhetővé, a felhasználói felületet kifejezővé tehetjük. A GTK programkönyvtár a fa képernyőelem kezelésére a **TreeView** típust biztosítja a programozó számára [2]. A fa képernyőelemet a 8.12. ábra mutatja be.



8.12. ábra. A fa képernyőelem

A fa képernyőelemen belül több oszlopot jeleníthetünk meg. Ehhez a GTK programkönyvtár a **GtkTreeViewColumn** típust biztosítja. A **GtkTreeViewColumn** megvalósítja a **GtkCellLayout** interfészt, így benne egy vagy több cellarajzoló is elhelyezhetünk. Ebből következik, hogy a fa képernyőelem egy oszlopában akár többféle adat is megjeleníthető, azaz a fa egy oszlopán belül több oszlop is lehet. Ezt figyelhetjük meg a 8.12. ábrán, ahol a fa első oszlopában egy kép és egy szöveg is megjelenik.

A fa képernyőelem használatát a következőképpen foglalhatjuk össze. A Glade segítségével el kell helyezni az ablakban a fa képernyőelemet és a létrehozását jelző jelzéshez (a **"realize"** jelzéshez) egy visszahívott függvényt kell kapcsolnunk, hogy a fában megjelenített adatokat és a fa képernyőelem oszlopait előkészíthessük. A visszahívott függvényben el kell készíteni a fa képernyőelem oszlopait a **GtkTreeViewColumn** adatszerkezeteket és mindegyikben el kell helyezni a megfelelő cellarajzókat.

Ezek után el kell készíteni a fában megjelenített adatokat tároló raktárat. Ennek kapcsán tudnunk kell, hogy a fa képernyőelem érzékeli, hogy lista vagy fa szerkezetű raktárat jelenítünk-e meg a segítségével. Ha a fa képernyőelem úgy érzékeli, hogy fa szerkezetű raktárat akarunk megjeleníteni, akkor az első oszlop előtt megfelelő méretű helyet hagy ki az egyes ágak nyitott és zárt állapotát jelző kis kpek számára. A képernyőelem a helyet akkor is kihagyja, ha a fa szerkezetű raktár történetesen egyetlen kinyitható ágot sem tartalmaz, azaz listaszerűen van

adatokkal feltöltve.

Ha létrehoztuk a raktárat és az adatokat megjelenítő cellarajzolókat is elhelyeztük a fa képernyőelem oszlopaiba, akkor a raktárat a képernyőelemhez rendelhetjük és ezzel tulajdonképpen használhatóvá is tesszük azt.

A fa képernyőelem kezelésére szolgáló függvények közül a legfontosabbak a következők.

`GtkTreeViewColumn *gtk_tree_view_column_new(void);` A függvény segítségével új oszlopot hozhatunk létre. A függvény visszatérési értéke az új oszlopot reprezentáló adatszerkezetet jelöli a memóriában.

`GtkTreeViewColumn *gtk_tree_view_column_new_with_attributes(const gchar *cím, GtkCellRenderer *cellarajzoló, ..., NULL);` A függvény segítségével új oszlopot hozhatunk létre, mégpedig úgy, hogy egyben beállítjuk a címét, az oszlopban megjelenő cellarajzolót és beállíthatjuk a cellarajzolóban megjelenő adatok oszlopának számát is.

A függvény első paramétere az oszlop címét megadó karakterlánc címe a memóriában. A fa képernyőelem fejlécében az oszlop felett ez a szöveg fog megjeleni a képernyőn.

A függvény második paramétere az oszlopban elhelyezendő cellarajzoló. Nyilvánvalóan akkor érdemes ezt az egyszerűsített függvényt használnunk az oszlop létrehozására, ha az oszlopban csak egy cellarajzolót akarunk elhelyezni.

A függvény további paramétere a cellarajzoló tulajdonságaiból és a hozzájuk rendelt oszlopszámból állnak. A párok megadják, hogy az oszlopban található cellarajzoló a fa képernyőelemhez rendelt raktár mely oszlopait jelenítse meg és milyen formában. A párokat megadó paraméterek után utolsó paraméterként a `NULL` értéket kell megadnunk.

A függvény visszatérési értéke az új oszlopot kezelő adatszerkezetet jelöli a memóriában.

`gint gtk_tree_view_append_column(GtkTreeView *fa, GtkTreeViewColumn *oszlop);` A függvény segítségével a fa képernyőelemhez új oszlopot adhatunk, ami a már elhelyezett oszlopoktól jobbra, utolsóként fog megjeleni a képernyőn.

A függvény első paramétere a fa képernyőelemet, második paramétere pedig az elhelyezendő oszlopot jelöli a memóriában. A függvény visszatérési értéke megadja, hogy az új oszlop elhelyezése után hány oszlop van a fában.

300

`void gtk_tree_view_set_model(GtkTreeView *fa, GtkTreeModel *raktár);` A függvény segítségével beállíthatjuk, hogy a fa képernyőelemben melyik raktár adatai jelenjenek meg.

A függvény első paramétere a beállítandó fát, második paramétere pedig a megjelenítendő raktárat jelöli a memóriában.

A következő példa bemutatja hogyan jeleníthetünk meg adatokat szöveges és képi formában egyaránt a fa képernyőelemben. A példa függvényei által létrehozott képernyő képét a 8.12. képen látható.

53. példa. A következő függvény az adatok megjelenítését végző oszlopokat és cellarajzolókat helyezi el a fa képernyőelemben.

```

1 static void
2 add_header_to_treeview_simple(GtkTreeView *tree_view)
3 {
4     GtkTreeViewColumn *column;
5     GtkCellRenderer *renderer;
6
7     column = gtk_tree_view_column_new();
8     gtk_tree_view_column_set_title(column, "Hely");
9     renderer = gtk_cell_renderer_pixbuf_new();
10    gtk_cell_layout_pack_start(GTK_CELL_LAYOUT(column),
11                               renderer, FALSE);
12    gtk_cell_layout_add_attribute(GTK_CELL_LAYOUT(column),
13                                  renderer, "icon-name", 0);
14    renderer = gtk_cell_renderer_text_new();
15    gtk_cell_layout_pack_start(GTK_CELL_LAYOUT(column),
16                               renderer, FALSE);
17    gtk_cell_layout_add_attribute(GTK_CELL_LAYOUT(column),
18                                  renderer, "text", 1);
19    renderer = gtk_cell_renderer_text_new();
20    gtk_tree_view_append_column(tree_view, column);
21
22    column = gtk_tree_view_column_new_with_attributes(
23        "Könyvtár", renderer,
24        "text", 2, NULL);
25    gtk_tree_view_append_column(tree_view, column);
26 }
```

A függvény két oszlopot hoz létre a paramétere által jelölt fa képernyőelemen belül.

Az első oszlop egy kép- és egy szöveges cellarajzolót tartalmaz, ezért a `gtk_tree_view_column_new()` függvénnyel hozzuk létre (7. sor) és

cellaelrendezésként kezelve elhelyezzük benne a két cellarajzolókat a 10–11. és a 15–16. sorokban. Az első oszlopot a 20. sorban helyezzük el a fe képernyőelemben.

A második oszlop csak egy szöveges cellarajzolókat tartalmaz, ezért az egyszerűsített `gtk_tree_view_column_new_with_attributes()` függvénnyel hozzuk létre a 22–24. sorokban. A második oszlopot a 25. sorban helyezzük el a fán belül.

A következő függvény a fa képernyőelem létrehozásakor, a *“realize”* jelzés hatására hívódik meg.

```

1 void
2 on_treeview1_realize(GtkWidget *widget,
3                       gpointer user_data)
4 {
5     GtkTreeStore *tree_store;
6     add_header_to_treeview(GTK_TREE_VIEW(widget));
7     tree_store = storage_tree_store_new();
8
9     gtk_tree_view_set_model(GTK_TREE_VIEW(widget),
10                             GTK_TREE_MODEL(tree_store));
11 }
```

A függvény a példa előző függvényét a 6. sorban hívja, hogy a képernyőelemben az oszlopokat és a cellarajzókat elhelyezze. Ezek után a 7. sorban hívjuk a fa szerkezetű raktárat létrehozó és adatokkal feltöltő függvényt, amelyet a rövidség kedvéért most nem mutatunk be.

Utolsó lépésként a 9–10. sorban az adatokkal felszerelt fa szerkezetű raktárat a fa képernyőelemhez rendeljük.

8.7.1. Az oszlopok finomhangolása

A következő néhány függvény segítségével a fa képernyőelem oszlopait finomhangolhatjuk, olyan megjelenésre és viselkedésre állíthatjuk, ami a legjobban megfelel az igényeinknek.

`void gtk_tree_view_column_set_spacing(GtkTreeViewColumn *oszlop, gint keret);` A függvény segítségével beállíthatjuk, hogy az oszlopban az egyes cellarajzók közt hány képpontnyi helyet akarunk kihagyni.

A függvény első paramétere az oszlopot jelzi a memóriában, a második paramétere pedig megadja, hogy a belsőben hány képpontnyi üres rész jelenjen meg az egyes cellarajzók közt.

`void gtk_tree_view_column_set_visible(GtkTreeViewColumn *oszlop, gboolean látható);` A függvény segítségével beállíthatjuk, hogy az adott oszlop látható legyen-e a képernyőn.

A függvény első paramétere az oszlopot jelzi a memóriában, a második paramétere pedig megadja, hogy az oszlop megjelenjen-e a képernyőn. Ha a második paraméter értéke `FALSE`, az oszlop nem látszik a képernyőn.

`void gtk_tree_view_column_set_resizable(GtkTreeViewColumn *oszlop, gboolean méretezhető);` A függvény segítségével beállíthatjuk, hogy az adott oszlop méretezhető legyen-e, azaz a felhasználó az oszlop méretét megváltoztathatja-e a fejlécen az oszlopok közti elválasztójel mozgásával.

A függvény első paramétere a fa képernyőelemet jelöli a memóriában, második paramétere pedig megadja, hogy az oszlop méretezhető legyen-e. Alapértelmezés szerint az oszlopok nem méretezhetők.

`void gtk_tree_view_column_set_sizing(GtkTreeViewColumn *oszlop, GtkTreeViewColumnSizing méretezés);` A függvény segítségével beállíthatjuk, hogy a fa képernyőelem adott oszlopának szélessége hogyan alakuljon a képernyőn.

A függvény első paramétere az oszlopot jelöli a memóriában, a második paramétere pedig a következő állandók egyike lehet:

`GTK_TREE_VIEW_COLUMN_GROW_ONLY` A beállított szélességhez képest a GTK programkönyvtár a szélességet legfeljebb növelheti, ha szükséges.

`GTK_TREE_VIEW_COLUMN_AUTOSIZE` Az oszlop szélességét a GTK programkönyvtár automatikusan állítja be.

`GTK_TREE_VIEW_COLUMN_FIXED` Az oszlop szélessége állandó.

`gint gtk_tree_view_column_get_width(GtkTreeViewColumn *oszlop);` A függvény segítségével az oszlop aktuális szélessége lekérdezhető.

A függvény paramétere a fa képernyőelem egy oszlopát jelöli, a visszatérési értéke pedig megadja, hogy az oszlop hány képpontnyi szélességű helyet foglal a képernyőn.

`void gtk_tree_view_column_set_fixed_width(GtkTreeViewColumn *oszlop, gint szélesség);` A függvény segítségével beállíthatjuk, hogy mekkora legyen az oszlop szélessége, ha az oszlopot állandó szélességgel jelenítjük meg.

A függvény első paramétere az oszlopot jelöli a memóriában, a második paramétere pedig a szélességet adja meg képpontban. A szélességnek 0-nál nagyobbnak kell lennie.

`void gtk_tree_view_column_set_min_width(GtkTreeViewColumn *oszlop, gint szélesség);` A függvény segítségével beállíthatjuk, hogy az oszlop legalább mekkora szélességgel jelenjen meg a képernyőn, ha a szélessége változhat.

A függvény első paramétere az oszlopot jelöli a memóriában, a második paramétere pedig megadja a lehető legkisebb szélességet. A második paraméter értéke lehet `-1` is, ami azt jelenti, hogy az oszlop tetszőlegesen kis szélességgel jelenhet meg.

`void gtk_tree_view_column_set_max_width(GtkTreeViewColumn *oszlop, gint szélesség);` A függvény segítségével megadhatjuk, hogy legfeljebb milyen szélességgel jelenjen meg az oszlop, ha az oszlop szélessége változó lehet.

A függvény első paramétere a beállítandó oszlopot jelöli a memóriában, a második paraméter pedig megadja az oszlop legnagyobb szélességét. Ha a második paraméter értéke `-1`, az oszlop szélessége tetszőlegesen nagy lehet.

`void gtk_tree_view_column_set_title(GtkTreeViewColumn *oszlop, const gchar *cím);` A függvény segítségével beállíthatjuk a fa képernyőelem oszlopának címét.

A függvény első paramétere az oszlopot, a második paramétere pedig az oszlop új címét jelöli a memóriában.

`void gtk_tree_view_column_set_expand(GtkTreeViewColumn *oszlop, gboolean nyújtt);` A függvény segítségével beállíthatjuk, hogy a fa képernyőelem oszlopa nyújtható legyen-e, azaz, ha a fa képernyőelem szélesebb a szükségesnél, akkor az adott oszlop szélességét növelni kell-e. Alapértelmezés szerint az oszlopok szélessége nem növekszik ilyen esetben, a „felesleges” szélességet a GTK programkönyvtár az utolsó oszlop szélességének növelésével használja fel.

A függvény első paramétere az oszlop címe, a második paramétere pedig logikai érték, ami megadja, hogy az oszlop szélességét növelni kell-e.

`void gtk_tree_view_column_set_clickable(GtkTreeViewColumn *oszlop, gboolean aktív);` A függvény segítségével megadhatjuk, hogy az adott oszlop fejlécbeli címe aktív legyen-e, azaz érzékelje-e, ha a felhasználó az egérrel rákattint.

304

A függvény első paramétere az oszlopot jelöli a memóriában, a második paramétere pedig logikai érték, ami megadja, hogy az oszlop fejléce érzékeny legyen-e.

`void gtk_tree_view_column_set_widget(GtkTreeViewColumn *oszlop, GtkWidget *elem);` A függvény segítségével tetszőleges képernyőelemet helyezhetünk el a fa képernyőelem oszlopának fejlécében. Alapértelmezett esetben a fejlécben az oszlop címét megjelenítő címke jelenik meg.

A függvény első paramétere az oszlopot, a második a fejlécben elhelyezendő képernyőelemet jelöli a memóriában.

`void gtk_tree_view_column_set_alignment(GtkTreeViewColumn *oszlop, gfloat xalign);` A függvény segítségével megadhatjuk, hogy az oszlop fejlécében található címke a fejléc melyik részén jelenjen meg.

A függvény első paramétere az oszlopot jelöli a memóriában, a második paramétere pedig egy 0,0 és 1,0 közé eső szám, ami megadja, hogy a címke a rendelkezésre álló terület mely részén jelenjen meg. A 0,0 érték hatására a címke bal oldalon, az 1,0 érték hatására jobb oldalon, a 0,5 érték hatására pedig középen jelenik meg.

`void gtk_tree_view_column_set_reorderable(GtkTreeViewColumn *oszlop, gboolean rendezhető);` A függvény segítségével megadhatjuk, hogy a fa képernyőelem adott oszlopa a többi oszlophoz képest elmozdítható legyen-e. Ha az összes oszlopot elmozdíthatóra állítjuk, a felhasználó az egér segítségével az oszlopok fejlécét vízszintesen elmozdíthatja, így az oszlopok sorrendjét tetszőlegesen módosíthatja.

A függvény első paramétere az oszlopot jelöli a memóriában, a második paramétere pedig egy logikai érték, ami megadja, hogy az oszlop elmozdítható legyen-e.

`void gtk_tree_view_column_set_sort_column_id(GtkTreeViewColumn *oszlop, gint azonosító);` A függvény segítségével megadhatjuk, hogy a fa képernyőelemben megjelenő raktár melyik oszlopa szerint kell rendezni a képernyőelem sorait, ha a felhasználó az adott oszlop fejlécére kattint. Ha ezt a tulajdonságot beállítjuk, az adott oszlop fejlécére kattintva a sorba rendezés automatikusan megtörténik.

A függvény első paramétere az oszlopot jelöli a memóriában, a második paramétere pedig megadja, hogy az adott oszlop aktiválása esetében melyik oszlop szerint kell rendezni a sorokat.

```
void gtk_tree_view_column_set_sort_indicator(GtkTreeViewColumn
*oszlop, gboolean megjelenik);
```

E függvény segítségével az oszlop fejlécében megjelenő, lefelé vagy felfelé mutató nyilat kapcsolhatjuk be. Ez a nyíl azt jelzi a felhasználónak, hogy a fa képernyőelem sorai az adott oszlop szerint vannak rendezve.

Ha egyszerűen csak azt akarjuk, hogy a felhasználó az oszlopokat a fejlécre kattintással rendezhesse, akkor erre a függvényre nincsen szükségünk, csak a `gtk_tree_view_column_set_sort_column_id()` függvényt kell használnunk.

Ha be akarjuk állítani a rendezést jelző nyíl irányát, az alább ismertetett `gtk_tree_view_column_set_sort_order()` függvényt kell használnunk.

A függvény első paramétere az oszlopot jelöli a memóriában, a második paramétere pedig megadja, hogy a rendezést jelző nyíl be legyen-e kapcsolva.

```
void gtk_tree_view_column_set_sort_order(GtkTreeViewColumn
*oszlop, GtkSortType sorrend);
```

A függvény segítségével beállíthatjuk, hogy a fejlécben a rendezést jelző nyíl milyen irányba mutasson. Ez a függvény a sorokat nem rendezi, egyszerűen csak beállítja, hogy melyik irányba mutasson a nyíl.

A függvény első paramétere az oszlopot jelöli a memóriában, a második paramétere pedig meghatározza a rendezést jelző nyíl irányát. Itt a következő állandók egyikét használhatjuk.

GTK_SORT_ASCENDING A nyíl növekvő sorrendet jelez, felfelé mutat.

GTK_SORT_DESCENDING A nyíl csökkenő sorrendet mutat, lefelé mutat.

A fa képernyőelem oszlopainak tulajdonságát a G programkönyvtár tulajdonságok kezelésére alkalmas eszközeivel – a `g_object_set()` és `g_object_get()` – is beállíthatjuk és lekérdezhethetjük. A tulajdonságokat a következő lista mutatja be, a rövidség kedvéért csak vázlatosan, hiszen a legtöbbjüket a függvények kapcsán már részletesen is ismertettük.

"alignment", **gfloat** Az oszlop fejlécében található címke rendezése.

"clickable", **gboolean** Logikai érték, ami megadja, hogy a fejléc érzékelje-e ha a felhasználó rákattint.

"expand", **gboolean** Logikai érték, ami megadja, hogy a „felesleges” szélességből az adott oszlop részesüljön-e.

306

- "fixed-width", gint** Az oszlop szélessége, ha az oszlop állandó szélességű.
- "max-width", gint** Az oszlop legnagyobb szélessége, ha az oszlop szélessége változó, -1 a tetszőlegesen nagy szélesség jelölésére.
- "min-width", gint** Az oszlop legkisebb szélessége, ha az oszlop szélessége változó, -1 a tetszőlegesen kicsi szélesség jelölésére.
- "reorderable", gboolean** Logikai érték, ami megadja, hogy az oszlop a többi oszlophoz képest áthelyezhető legyen-e.
- "resizable", gboolean** Logikai érték, ami megadja, hogy a felhasználó megváltoztathassa-e az oszlop szélességét.
- "sizing", GtkTreeViewColumnSizing** A szélesség automatikus megállapításának módja, ami lehet **GTK_TREE_VIEW_COLUMN_GROW_ONLY**, **GTK_TREE_VIEW_COLUMN_AUTOSIZE**, vagy **GTK_TREE_VIEW_COLUMN_FIXED**.
- "sort-indicator", gboolean** Logikai érték, ami megadja, hogy fejlécben megjelenjen-e a rendezést jelző nyíl.
- "sort-order", GtkSortType** A fejlécben található nyíl iránya, ami lehet **GTK_SORT_ASCENDING** vagy **GTK_SORT_DESCENDING**.
- "spacing", gint** Megadja, hogy az oszlopon belül elhelyezett cellarajzók között hány képpontnyi szélességű üres helyet kell megjeleníteni.
- "title", gchararray** Az oszlop címe, ami alapértelmezés szerint a fejlécben elhelyezett címkében is megjelenik.
- "visible", gboolean** Logikai érték, ami megadja, hogy a fa képernyőelem adott oszlopa látszik-e a képernyőn vagy rejtett.
- "widget", GtkWidget *** A fejlécben megjelenő képernyőelem címe. Alapértelmezett esetben a fejlécben egy címke jelenik meg az oszlop címével.
- "width", gint** Csak olvasható tulajdonság, ami megadja, hogy az oszlop szélessége az adott pillanatban hány képpontnyi.

Szerencsés, ha az alkalmazásban található összes fa összes oszlopa hasonlóképpen működik és így a felhasználót a legkevesebb meglepetés éri. Ezt legkönnyebben úgy érhetjük el, ha az alkalmazás minden fa képernyőelemének minden oszlopát ugyanazzal a függvénnyel hozzuk létre. Ezt mutatja be a következő példa.

54. példa. A következő sorok egy egyszerű függvényt mutatnak be, amelyek a fa képernyőelemhez új oszlopot ad hozzá. Az új oszlop egy szöveges cellarajzoló is tartalmaz az adatok megjelenítésére.

```

1  GtkTreeViewColumn *
2  tree_view_column_text_create(
3      GtkTreeView *tree_view,
4      gint         column_number,
5      gchar        *title)
6  {
7      GtkTreeViewColumn *column;
8      GtkCellRenderer  *renderer;
9      /*
10     * Az oszlop.
11     */
12     column = gtk_tree_view_column_new();
13     gtk_tree_view_append_column(tree_view, column);
14     gtk_tree_view_column_set_sort_column_id(column,
15         column_number);
16     g_object_set(G_OBJECT(column),
17         "title", title,
18         "clickable", TRUE,
19         "reorderable", TRUE,
20         "resizable", TRUE,
21         "expand", TRUE,
22         NULL);
23     /*
24     * A cellarajzoló.
25     */
26     renderer = gtk_cell_renderer_text_new();
27     gtk_tree_view_column_pack_start(column, renderer,
28         TRUE);
29     gtk_tree_view_column_add_attribute(column, renderer,
30         "text", column_number);
31     return column;
32 }

```

A függvény első paramétere a fa képernyőelemet jelöli a memóriában. A második paraméter megadja, hogy a létrehozandó oszlop a fához tartozó raktár hányadik oszlopát jelenítse meg. A függvény harmadik paramétere az új oszlop címét megadó karakterláncot jelöli a memóriában.

8.7.2. A fa képernyőelem beállításai

A fa képernyőelem megjelenése és viselkedése jónéhány függvény és tulajdonság segítségével finomhangolható. A következőkben először a finomhangolásra használható függvényeket vesszük sorra.

`void gtk_tree_view_set_enable_search(GtkTreeView *fa, gboolean gyorskeresés);` A függvény segítségével beállíthatjuk, hogy a fa képernyőelem támogassa-e a szavak begépelésén alapuló gyorskeresést. Ha a gyorskeresését bekapcsoljuk, a felhasználó egyszerűen elkezdheti begépelni a keresett szöveget, a fa pedig minden billentyűleütés után a legjobb találathoz ugrik.

A függvény első paramétere a képernyőelemet jelöli a memóriában, második paramétere pedig egy logikai érték, ami megadja, hogy támogatjuk-e a begépelésen alapuló gyorskeresést.

`void gtk_tree_view_set_enable_tree_lines(GtkTreeView *fa, gboolean vonalak);` A függvény segítségével beállíthatjuk, hogy a fában a leszármazottakat vonalak jelöljék-e.

A függvény első paramétere a képernyőelemet jelöli a memóriában, a második paramétere pedig egy logikai érték, ami megadja, hogy a vonalakat meg kívánjuk-e jeleníteni.

`void gtk_tree_view_set_fixed_height_mode(GtkTreeView *fa, gboolean egységes);` A függvény segítségével felgyorsíthatjuk a fa képernyőelem működését a sorok magasságának egységesre állításával. Csak akkor használhatjuk azonban ezt a függvényt, ha a fa képernyőelem valóban azonos magasságú sorokat tartalmaz és az oszlopok szélessége is állandó.

A függvény első paramétere a fa képernyőelemet jelöli a memóriában, a második paramétere pedig egy logikai érték, ami megadja, hogy egységes sormagasságot akarunk-e használni.

`void gtk_tree_view_set_grid_lines(GtkTreeView *fa, GtkTreeViewGridLines hely);` A függvény segítségével beállíthatjuk, hogy a fa képernyőelemben a sorokat és az oszlopokat elválassa-e egy szaggatott vonal.

A függvény első paramétere a fa képernyőelemet jelöli a memóriában. A második paraméter megadja, hogy hol jelenjenek meg szaggatott vonalak. Itt a következő állandók egyikét használhatjuk:

GTK_TREE_VIEW_GRID_LINES_NONE A szaggatott vonalak megjelenítésének tiltása.

GTK_TREE_VIEW_GRID_LINES_HORIZONTAL Csak vízszintes vonalak jelennek meg.

GTK_TREE_VIEW_GRID_LINES_VERTICAL Csak függőleges vonalak jelennek meg.

GTK_TREE_VIEW_GRID_LINES_BOTH Mind vízszintes, mind pedig függőleges vonalak is megjelennek.

void gtk_tree_view_set_headers_clickable(GtkTreeView *fa, gboolean érzékeny); A függvény segítségével egy lépésben beállíthatjuk, hogy az oszlopok fejlécei érzékenyek legyenek-e a felhasználó kattintására. Ezt a tulajdonságot oszloponként is beállíthatjuk.

A függvény első paramétere a fa képernyőelemet jelöli a memóriában, a második paramétere pedig logikai érték, ami meghatározza, hogy a fejléc oszlopokat jelző részei érzékenyek legyenek-e.

void gtk_tree_view_set_headers_visible(GtkTreeView *fa, gboolean látható); A függvény segítségével beállíthatjuk, hogy a fejléc megjelenjen-e a fa képernyőelem felső részén.

A függvény első paramétere a képernyőelemet jelöli a memóriában, a második paramétere pedig logikai érték, ami meghatározza, hogy a képernyőelem felső részén megjelenjen-e a fejléc.

void gtk_tree_view_set_hover_expand(GtkTreeView *fa, gboolean automatikus_nyitás); A függvény segítségével beállíthatjuk, hogy használni akarjuk-e a fában megjelenített ágak automatikus kinyitását. Ha ezt a lehetőséget használjuk a felhasználónak nem kell lenyomnia az egér gombját az egyes ágak kinyitásához, elég ha egyszerűen a kinyitandó ág fölé áll az egérrel.

A függvény első paramétere a fát jelöli a memóriában, a második paramétere pedig logikai érték, ami meghatározza, hogy használjuk-e az automatikus nyitás lehetőségét.

void gtk_tree_view_set_hover_selection(GtkTreeView *fa, gboolean automatikus_kijelölés); A függvény segítségével megadhatjuk, hogy használni akarjuk-e az automatikus kijelölést a fában. Ha élünk ezzel a lehetőséggel, akkor a felhasználónak nem kell lenyomnia az egér gombját ahhoz, hogy az adott sort kijelölje, elég, ha egyszerűen a sor fölé áll az egérrel. Ezt a lehetőséget csak olyan fák esetében használhatjuk, ahol egyszerre csak egy sor lehet kijelölve.

A függvény első paramétere a képernyőelemet jelöli a memóriában, a második paramétere pedig megadja, hogy használni akarjuk-e az automatikus kijelölést.

310

`void gtk_tree_view_set_reorderable(GtkTreeView *fa, gboolean átrakható);` A függvény segítségével beállíthatjuk, hogy engedélyezni akarjuk-e a felhasználó számára, hogy a fa képernyőelemben megjelenő sorokat az egérrel átrendezze. Ha engedélyezzük ezt a lehetőséget, a felhasználó a lista szerkezetű raktár elemeinek sorrendjét megváltoztathatja, a fa szerkezetű raktárban pedig akár az elemek leszármazási rendjét is módosíthatja.

A felhasználó által kezdeményezett módosítás során a fa képernyőelem valóban átrendezi a raktár tartalmát. Ha erről azonnal értesülni akarunk, akkor a raktár `"row_inserted"` (sor beszúrása megtörtént) és a `"row_deleted"` (sor törlése megtörtént) jelzésekhez kell a megfelelő típusú visszahívott függvényt rendelnünk.

A függvény első paramétere a fa képernyőelemet jelöli a memóriában, a második paramétere pedig egy logikai érték, ami megadja, hogy a sorok átrendezhetők legyenek-e.

`void gtk_tree_view_set_rules_hint(GtkTreeView *fa, gboolean színez);` A függvény segítségével beállíthatjuk, hogy az egy sorhoz tartozó oszlopok azonosítását megkönnyítő színezéssel akarjuk-e megjeleníteni az adatokat. Ha ezt a lehetőséget használjuk a fa egyes sorai enyhén eltérő háttérszínnel jelennek meg, hogy az olvasó könnyebben követhesse a sorokat az olvasás során.

A függvény első paramétere a fa képernyőelemet jelöli a memóriában, a második paramétere pedig megadja, hogy használni kívánjuk-e a színezést.

`void gtk_tree_view_set_search_column(GtkTreeView *fa, gint oszlop);` A függvény segítségével megadhatjuk, hogy a gyorskeresés üzemmódban a keresés a fa képernyőelemhez tartozó raktár melyik oszlopában történjen.

A függvény első paramétere a fa képernyőelemet jelöli a memóriában, a második paramétere pedig megadja, hogy a gyorskereséskor a raktár hányadik oszlopában kell keresni. Az oszlopok számozása 0-tól indul.

`void gtk_tree_view_set_search_entry(GtkTreeView *fa, GtkEntry *beviteli_mező);` A függvény segítségével megadhatjuk, hogy a fa a gyorskeresésre melyik beviteli mezőt használja. Alapértelmezett esetben a fa egy beépített beviteli mezőt használ, ami csak a keresés ideje alatt jelenik meg.

A függvény első paramétere a fa képernyőelemet, a második paramétere pedig a használni kívánt beviteli mezőt jelöli a memóriában. Ha a második paraméter értéke `NULL`, a fa újra a beépített beviteli mezőt kezdi használni.

8.7.3. A fa képernyőelem használata

A fa képernyőelem kapcsán mindenképpen említésre érdemesek azok az eszközök, amelyek segítségével a fa képernyőelem állapotát, tartalmát lekérdezhetjük. Az ilyen célokra használható függvényeket mutatjuk be a következő oldalakon. Nyilvánvaló, hogy elsősorban nem a fa képernyőelem G programkönyvtár által kezelt tulajdonságairól lesz szó – hiszen azokat általában mi magunk állítunk be és ezek különben is egyszerűen lekérdezhetőek a `g_object_get()` függvény segítségével – hanem a felhasználó által módosítható állapotról és tartalomról. Szintén fontosak azok a függvények, amelyek segítségével a fa képernyőelemet használat közben vezérelhetjük. A lekérdezésre szolgáló függvények mellett azokat a függvényeket is megtaláljuk a következő oldalakon, amelyekkel a működést „menet közben” befolyásolhatjuk.

A munka során nyilvánvalóan igen fontos, hogy lekérdezhessük a fa képernyőelemben megjelenített raktár címét. Erre használhatjuk a következő függvényt.

```
GtkTreeModel *gtk_tree_view_get_model(GtkTreeView *fa);
```

A függvény segítségével lekérdezhetjük a raktár címét, amelynek az adatait az adott pillanatban a fa képernyőelem megjeleníti.

A függvény paramétere a fa képernyőelemet jelöli a memóriában, a visszatérési értéke pedig a fában megjelenő raktár címét adja meg, ha a fához rendeltünk raktárat vagy `NULL` érték, ha nem.

Ha a visszatérési értékként kapott címről meg akarjuk tudni, hogy az lista szerkezetű vagy fa szerkezetű raktárat jelöl-e a memóriában, akkor használjuk a `GTK_IS_LIST_STORE()` és a `GTK_IS_TREE_STORE()` makrókat.

Ha be akarjuk állítani, hogy a fában melyik raktár tartalmát akarjuk megjeleníteni, akkor a már bemutatott `gtk_tree_view_set_model()` függvényt kell használnunk.

Természetesen nagyon fontos az is, hogy a felhasználó a fa képernyőelemen belül megjelenő sorok közül melyiket választotta ki. A GTK programkönyvtár a fa képernyőelemhez a kijelölés kezelésére külön típust, a `GtkTreeSelection` típust használja. A `GtkTreeSelection` típus kezelésére szolgáló függvények közül a legfontosabbak a következők.

```
GtkTreeSelection *gtk_tree_view_get_selection(GtkTreeView *fa);
```

E függvény segítségével lekérdezhetjük a fa képernyőelemhez tartozó, a kijelölést kezelő adatszerkezet címét.

A függvény paramétere a fa képernyőelemet jelöli a memóriában, a visszatérési értéke pedig megadja hol találjuk a kijelölést kezelő adatszerkezetet.

`void gtk_tree_selection_set_mode(GtkTreeSelection *választás, GtkSelectionMode mód);` A függvény segítségével beállíthatjuk, hogy a fa képernyőelem milyen módon tegye lehetővé a sorok kijelölését a felhasználó számára.

A függvény első paramétere a kijelölést kezelő adatszerkezetet jelöli a memóriában, a második paramétere pedig a következő állandók egyike lehet:

GTK_SELECTION_NONE A fában a felhasználó egyetlen sort sem jelölhet ki.

GTK_SELECTION_SINGLE A felhasználó legfeljebb egy sort jelölhet ki, így a kijelölt sorok száma mindig 0 vagy 1 lesz.

GTK_SELECTION_BROWSE A fában a felhasználó egy sort jelölhet ki. Ha úgy jelenítjük meg a fát, hogy előtte egyetlen sort sem jelöltünk ki, akkor nem lesz kijelölt sor, de a felhasználó az általa végzett kijelölést csak úgy vonhatja vissza, hogy egy másik sort jelöl ki. Így, ha a fa megjelenítése előtt kijelölünk egy sort, akkor a fában mindig pontosan egy kijelölt sor lesz.

A fa képernyőelemnek ez az alapértelmezett kijelölési módja.

GTK_SELECTION_MULTIPLE Ebben az üzemmódban tetszőleges számú sor kijelölhető.

Ebben az üzemmódban a felhasználó új elemet jelölhet ki anélkül, hogy a már kijelölt elemek kijelölését visszavonná, ha lenyomja a Ctrl billentyűt, sőt a sorok egész tartománya is kijelölhető a Shift billentyű segítségével.

`GtkSelectionMode gtk_tree_selection_get_mode(GtkTreeSelection *választás);` A függvény segítségével lekérdezhethetjük, hogy a fa képernyőelem milyen módon teszi lehetővé a felhasználónak a sorok kijelölését.

A függvény paramétere a kijelölést kezelő adatszerkezetet jelöli a memóriában, a visszatérési értéke pedig az előző – a `gtk_tree_selection_set_mode()` – függvénynél bemutatott állandók egyike.

`gboolean gtk_tree_selection_get_selected(GtkTreeSelection *választás, GtkTreeModel **raktár, GtkTreeIter *bejáró);` A függvény segítségével lekérdezhethetjük, hogy a fa képernyőelemen van-e kijelölt elem és ha van, akkor a fához tartozó raktár melyik sora van kijelölve. Ezt a függvényt kizárólag akkor használhatjuk, ha a fa egy időben legfeljebb csak egy sor kijelölését teszi lehetővé, azaz, ha a kijelölési módja **GTK_SELECTION_SINGLE** vagy **GTK_SELECTION_BROWSE**.

A függvény első paramétere a kijelölést kezelő adatszerkezetet jelöli a memóriában. A második paramétere annak a mutatónak a helyét jelöli, ahová a fa képernyőelemhez tartozó raktár címét el akarjuk helyezni. Ennek a paraméternek az értéke lehet `NULL` is, ekkor a függvény a raktár címét nem adja meg a függvény.

A harmadik paraméter egy bejáró címét jelöli a memóriában. A függvény ezt a bejárót a kijelölt sorra állítja. A harmadik paraméter is lehet `NULL`, ekkor a függvény nem adja meg, hogy melyik elem van kijelölve, csak a visszatérési értékét használhatjuk fel.

A függvény visszatérési értéke megadja, hogy van-e kijelölt elem, azaz igaz logikai értékű, ha a felhasználó kiválasztott egy elemet a képernyőn.

GList *gtk_tree_selection_get_selected_rows(GtkTreeSelection *választás, GtkTreeModel **raktár); A függvény segítségével lekérdezhethetjük a fa képernyőelemen kijelölt sorok listáját. A függvény egy egy listát hoz létre, amelyben `GtkTreePath` típusú elemek vannak, mindegyikük egy kijelölt sort jelölve a fához tartozó raktárban.

A függvény első paramétere a kijelölést kezelő adatszerkezetre mutat, a második paraméter pedig azt a mutatót jelöli a memóriában, ahova a fa képernyőelemhez tartozó raktár címét akarjuk elhelyezni. Ez utóbbi paraméter értéke lehet `NULL` is.

A függvény visszatérési értéke a függvény által létrehozott egyszerűen láncolt listát jelöl a memóriában. A lista elemei `GtkTreePath` típusú adatszerkezeteket jelölnek a memóriában.

Ha a listát meg akarjuk semmisíteni, minden `GtkTreePath` típusú elemet fel kell szabadítanunk a `gtk_tree_path_free()` függvénnyel, majd a listát is meg kell semmisítenünk a `g_list_free()` függvénnyel.

gint gtk_tree_selection_count_selected_rows(GtkTreeSelection *választás); A függvény segítségével lekérdezhethetjük, hogy hány sor van kijelölve a fa képernyőelemen.

A függvény paramétere a kijelölést kezelő adatszerkezetet jelöli a memóriában, a visszatérési értéke pedig megadja, hogy az adott pillanatban hány sor van kijelölve.

void gtk_tree_selection_select_path(GtkTreeSelection *választás, GtkTreePath *ösvény); A függvény segítségével az ösvénnyel jelzett elemet kijelölhetjük.

314

A függvény első paramétere a kijelölést kezelő adatszerkezetet, a második paramétere pedig a kijelölendő sor helyét meghatározó ösvényt jelöli a memóriában.

void gtk_tree_selection_unselect_path(GtkTreeSelection *választás, GtkTreePath *ösvény); A függvény segítségével a képernyőelem adott során a kijelölést megszüntethetjük. Ha a sor nincs kijelölve a függvény nem változtatja meg a képernyőelem állapotát.

A függvény első paramétere a kijelölést kezelő adatszerkezetet, a második paramétere pedig a sort meghatározó ösvényt jelöli a memóriában.

gboolean gtk_tree_selection_path_is_selected(GtkTreeSelection *választás, GtkTreePath *ösvény); A függvény segítségével lekérdezhethetjük, hogy az adott ösvény által kijelölt sor ki van-e jelölve.

A függvény első paramétere a kijelölést kezelő adatszerkezetet, a második paramétere pedig az ösvényt jelöli a memóriában. A függvény visszatérési értéke igaz, ha az adott sor ki van jelölve.

void gtk_tree_selection_select_iter(GtkTreeSelection *választás, GtkTreeIter *bejáró); A függvény segítségével a bejáróval jelölt sort kijelölhetjük éppen úgy, mintha a felhasználó jelölte volna ki azt.

A függvény első paramétere a kijelölés kezelésére használt adatszerkezetet, a második paramétere pedig a bejárót jelöli a memóriában.

void gtk_tree_selection_unselect_iter(GtkTreeSelection *választás, GtkTreeIter *bejáró); A függvény segítségével a fa képernyőelemhez tartozó raktárban a bejáróval jelölt sor kijelölését a fában visszavonhatjuk.

A függvény első paramétere a kijelölést kezelő adatszerkezetet, a másik paramétere pedig a bejárót jelöli a memóriában.

gboolean gtk_tree_selection_iter_is_selected(GtkTreeSelection *választás, GtkTreeIter *bejáró); A függvény segítségével lekérdezhethetjük, hogy a bejáróval jelzett sor ki van-e jelölve.

A függvény első paramétere a fa képernyőelemhez tartozó, a kijelölés kezelésére használt adatszerkezetet jelöli a memóriában. A második paraméter a fa képernyőelemhez tartozó raktár egy sorát jelöli, a visszatérési érték pedig megadja, hogy az adott sor ki van-e jelölve.

`void gtk_tree_selection_select_all(GtkTreeSelection *választás);` A függvény segítségével a fa képernyőelem minden sorát kijelölhetjük egy lépésben.

A függvény paramétere a fa képernyőelemhez tartozó, a kijelölt elemek nyilvántartására szolgáló adatszerkezetet jelöli a memóriában.

`void gtk_tree_selection_unselect_all(GtkTreeSelection *választás);` A függvény segítségével a fa képernyőelemen található összes sor kijelölését megszüntethetjük.

A függvény paramétere a kijelölt elemek nyilvántartására szolgáló adatszerkezetet jelöli a memóriában.

`void gtk_tree_selection_select_range(GtkTreeSelection *választás, GtkTreePath *eleje, GtkTreePath *vége);` A függvény segítségével a fa képernyőelem egymás után következő sorait, a sorok egy tartományát jelölhetjük ki. Ehhez természetesen a fában először engedélyeznünk kell a többszörös kijelölést.

A függvény első paramétere a kijelölés kezelésére használt adatszerkezetet jelöli a memóriában. A második paraméter az első kijelölt sort, a harmadik paraméter pedig az utolsó kijelölt sort azonosító bejárót jelölő mutató.

`void gtk_tree_selection_unselect_range(GtkTreeSelection *választás, GtkTreePath *eleje, GtkTreePath *vége);` A függvény segítségével a fa képernyőelemen található sorok egy tartományán szüntethetjük meg a kijelölést.

A függvény paramétereinek jelentése megegyezik a `gtk_tree_selection_select_range()` függvény paramétereinek a jelentésével.

A következő oldalakon a fa képernyőelemen található kijelöléseket kezelő eszközök használatát mutatjuk be példákon keresztül.

55. példa. A következő igen egyszerű függvény úgy állítja be a fa képernyőelemet, hogy abban a felhasználó egy időben több sort is kijelölhessen.

```
1 inline void
2 w_tree_view_set_multiselect(GtkTreeView *tree_view)
3 {
4     GtkTreeSelection *selection;
5
6     selection = gtk_tree_view_get_selection(tree_view);
7     gtk_tree_selection_set_mode(selection,
```


316

```
8      GTK_SELECTION_MULTIPLE);
9  }
```

A függvény igen egyszerű, az előző oldalakon tárgyalt ismeretek alapján könnyen megérthető.

56. példa. A következő függvény bemutatja, hogy hogyan kérdezhetjük le és állíthatjuk be a fa képernyőelemen kijelölt sorok listáját.

```
1  gboolean
2  commissions_tree_view_refresh(GtkTreeView *tree_view)
3  {
4      GtkTreeModel      *tree_model;
5      GtkTreeSelection  *selection;
6      GList              *selected, *li;
7      GtkTreePath        *path;
8
9      tree_model = gtk_tree_view_get_model(tree_view);
10     g_assert(GTK_IS_TREE_STORE(tree_model));
11
12     /*
13      * A kijelölt elemek listájának mentése.
14      */
15     selection = gtk_tree_view_get_selection(tree_view);
16     selected = gtk_tree_selection_get_selected_rows(
17         selection, NULL);
18
19     /*
20      * A lista újraolvasása.
21      */
22     gtk_tree_store_clear(GTK_TREE_STORE(tree_model));
23     sql_fill_tree_store_list(GTK_TREE_STORE(tree_model),
24         connection, "SELECT * FROM commission_view;");
25
26     if (selected == NULL)
27         return TRUE;
28     /*
29      * A megfelelő elemek kijelölése.
30      */
31     li = selected;
32     while (li != NULL) {
33         path = li->data;
34         gtk_tree_selection_select_path(selection, path);
35         li = li->next;
```

```

36     }
37     /*
38      * A lista felszabadítása.
39      */
40     g_list_foreach(selected, gtk_tree_path_free, NULL);
41     g_list_free(selected);
42     return TRUE;
43 }
```

A függvény menti a kijelölt sorok listáját, újraolvassa a sorokat, majd újra kijelöli azokat a sorokat, amelyek a frissítés előtt ki voltak jelölve. Nyilvánvaló, hogy ha a frissítés során új sorok jelennek meg vagy csökken a sorok száma, akkor nem biztos, hogy célszerű az ugyanazon helyen megjelenő sorokat újra kijelölni, így összetettebb függvény segítségével logikusabban működő programot készíthetünk.

A függvény a 15–17. sorban lekérdezi a kijelölt sorokra mutató ösvények listáját a `gtk_tree_selection_get_selected_rows()` függvény segítségével, majd a 22. sorban törli a raktárat, hogy a 23–24. sorban újraolvassa.

A függvény ezek után a 31–36. sorok közt végigjárja a frissítés előtt kijelölt sorokat tartalmazó listát és a 34. sorban újra kijelöli a sorokat.

Ezek után már csak a lista felszabadítás van hátra, amelyet a 40–41. sorokban olvashatunk.

A fa képernyőelemen a sorok kijelöléséhez hasonlóan hasonlóan mozgathatunk az oszlopok között is. Ha az egyes oszlopokban található cellarajzolók szerkeszthetők, akkor a GTK programkönyvtár nem csak azt jelzi, hogy melyik sor van kijelölve, hanem azt is, hogy a soron belül melyik az aktív oszlop. A fa képernyőelemen belül van egy kurzor, amelyet a felhasználó a kurzormozgató billentyűkkel és az egérrel is mozgathat és amelynek megjelenése függ a beállított stílustól. A kurzor helyének lekérdezésére és beállítására a következő függvényeket használhatjuk.

```
void gtk_tree_view_get_cursor(GtkTreeView *fa,
    GtkTreePath **ösvény, GtkTreeViewColumn **oszlop);
```

A függvény segítségével lekérdezhethetjük, hogy a fa képernyőelemen a kurzor melyik sorban és melyik oszlopban található.

A függvény első paramétere a fa képernyőelemet jelöli a memóriában. A második és a harmadik paraméter egy-egy mutatót jelöl a memóriában, ahová a függvény az aktív sort jelölő ösvényt és az aktív oszlop címét elhelyezi. Ha egyetlen sor vagy oszlop sem, akkor a függvény a második, illetve a harmadik paraméterek által jelölt memóriaterületekre `NULL` értéket helyez.

Ha a függvény a második paraméterével jelzett memóriaterületre a `NULL` értéktől különböző értéket másol, akkor az egy újonnan létrehozott ösvényt jelöl a memóriában, amelyet használat után a `gtk_tree_path_free()` függvénnyel fel kell szabadítanunk.

```
void gtk_tree_view_set_cursor(GtkTreeView *fa,
    GtkTreePath *ösvény, GtkTreeViewColumn *oszlop,
    gboolean szerkesztés);
```

A függvény segítségével a fa képernyőelem aktív celláját állíthatjuk be.

A függvény első paramétere az ösvényt jelzi, ami meghatározza, hogy a kurzor melyik sorba kerüljön. A második paraméter a fa képernyőelem egy oszlopát jelöli és így meghatározza, hogy a kurzor melyik oszlopba kerüljön.

A függvény harmadik paramétere meghatározza, hogy a kurzor áthelyezése után az aktív cellát szerkeszteni akarjuk-e. Ha ennek a paraméternek igaz az értéke és a megfelelő cellarajzoló szerkeszthető, akkor a kurzor áthelyezése után a cella, amelyre a kurzort helyeztük azonnal szerkeszthető.

A kijelölt elemek programból való beállításához hasonlóan fontos, hogy szabályozni tudjuk a fa mely ágai legyenek nyitott állapotban. Erre a következő függvényeket használhatjuk.

```
void gtk_tree_view_expand_all(GtkTreeView *fa);
```

E függvény segítségével a fa képernyőelem összes ágát kinyithatjuk.

```
void gtk_tree_view_collapse_all(GtkTreeView *fa);
```

E függvény segítségével a fa képernyőelem összes ágát bezárhatjuk.

```
void gtk_tree_view_expand_to_path(GtkTreeView *fa,
    GtkTreePath *ösvény);
```

A függvény segítségével a fa képernyőelem egy adott sorának összes szülőjét kinyithatjuk, hogy a fa adott sora megjeleníthető legyen.

A függvény első paramétere a fa képernyőelemet jelöli a memóriában, a második paramétere pedig azt ösvényt jelzi, ami a kinyitandó sort jelöli.

```
gboolean gtk_tree_view_expand_row(GtkTreeView *fa,
    GtkTreePath *ösvény, gboolean mindet);
```

A függvény segítségével a fa képernyőelem adott ágát nyithatjuk ki.

A függvény első paramétere a fa képernyőelemet, a második paramétere a sort meghatározó ösvényt jelöli a memóriában. A függvény harmadik paramétere megadja, hogy csak az adott sor közvetlen leszármazottait akarjuk megjeleníteni vagy az összes közvetett leszármazottját is ki akarjuk nyitni. Ha a harmadik paraméter értéke

`TRUE` a függvény a sor összes közvetlen és közvetett leszármazottját is kinyitja.

A függvény visszatérési értéke igaz, ha a sor kinyitása lehetséges volt, azaz a sor és a leszármazottai is léteztek.

`gboolean gtk_tree_view_collapse_row(GtkTreeView *fa, GtkTreePath *ösvény);` A függvény segítségével a fa képernyőelem adott sorát bezárhatjuk.

A függvény első paramétere a fa képernyőelemet, második paramétere pedig a bezárandó sor ösvényét jelöli. A függvény visszatérési értéke igaz, ha a művelet sikeres volt.

A fa képernyőelem használata közben szükségünk lehet az oszlopok lekérdezésére. A következő néhány függvény segítségével a már létrehozott fa képernyőelem oszlopainak címét kérdezhetjük le kétféleképpen is.

`GtkTreeViewColumn *gtk_tree_view_get_column(GtkTreeView *fa, gint oszlopszám);` A függvény segítségével lekérdezhetjük a fa képernyőelem adott sorszámú oszlopjának címét. A függvény az oszlopok számozásakor a rejtett – a képernyőn nem látható – oszlopokat sem hagyja ki.

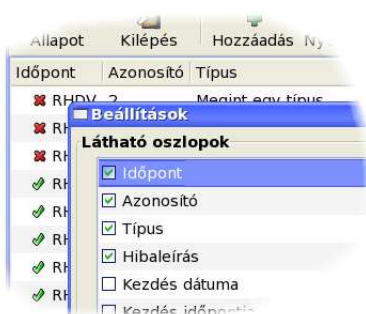
A függvény első paramétere a fa képernyőelemet jelöli a memóriában, a második paramétere pedig megadja, hogy hányadik oszlop címét akarjuk megtudni. Az oszlopok számozása 0-tól indul.

A függvény visszatérési értéke a paraméterrel jelzett oszlopot jelzi a memóriában ha a fában az adott sorszámú oszlop megtalálható és `NULL` ha nem.

`GList *gtk_tree_view_get_columns(GtkTreeView *fa);` A függvény segítségével a fa képernyőelemen található összes oszlop címét kérdezhetjük le egy lépésben.

A függvény paramétere a fát jelöli a memóriában. A visszatérési érték egy a függvény által létrehozott egyszeresen láncolt listát jelöl a memóriában. A lista minden eleme egy, a fa képernyőelem oszlopjának kezelésére szolgáló `GtkTreeViewColumn` típusú adatstruktúrát jelöl a memóriában. Ha a fát meg akarjuk semmisíteni csak a fa tárolására szolgáló memóriát kell felszabadítanunk a `g_list_free()` függvény segítségével.

A bemutatott függvények segítségével a képernyőn megjelenített fa képernyőelemek oszlopaikat elérhetjük. Az oszlopok címének segítségével az oszlop összes tulajdonságát lekérdezhetjük, vagy akár meg is változtathatjuk. Ennek a lehetőségnek a segítségével olyan függvényeket is készíthetünk, amelyekkel az alkalmazásunk összes fa és lista szerű képernyőelemének beállítását lehetővé teszi.



8.13. ábra. A fa képernyőelem oszlopainak kiolvasása

Ilyen eszközt láthatunk a 8.13. ábrán, ahol olyan ablakot mutatunk be, amelyben a felhasználó az oszlopok megjelenítését kapcsolhatja ki- és be. Az ablakban egy fa képernyőelem található, amiben a beállítandó fa oszlopnevei látszanak, soronként egy. A megvalósításhoz tehát olyan programot kell készítenünk, ami a beállítandó fa képernyőelem oszlopainak tulajdonságait egy másik, a beállításra használt fa soraiban helyezi. A beállításra használt fa soraiban minden oszlopnev előtt egy kapcsoló cellarajzoló is megjelenik, ami azt szabályozza, hogy a beállítandó fa adott oszlopa megjelenjen-e a képernyőn. A megvalósítás során tehát nem csak az egyes

oszlopok neveit kell kiolvasnunk, hanem azt is, hogy az adott oszlop megjelenjen-e a képernyőn, azaz látható-e vagy rejtett.

Nyilvánvaló, hogy miután a felhasználó elvégezte a beállítást, az egyes sorok értékeit felhasználva a beállítandó fa oszlopainak tulajdonságait be kell állítanunk.

Kérdés még természetesen az is, hogy a fa tulajdonságait beállító ablakot mikor jelenítjük meg a képernyőn és hogyan adjuk át neki a beállítandó fa memóriabeli címét. Erre a feladatra a legalkalmasabbnak – a felhasználó szempontjából legtermészetesebbnek – a fa területén megjeleníthető felbukkanó menü tűnik. A fa képernyőelemén előhívott felbukkanó menük kezelésére a 324. oldalon található 58 példában térünk mutatunk be egy lehetséges megvalósítást.

A 8.13. ábrán látható beállítóablak függvényeinek fontosabb részeit a következő példa mutatja be.

57. példa. A következő programrészlet egy fa képernyőelem (*to_set*) oszlopainak memóriacímét kérdezi le, majd a címek felhasználásával kiolvassa az oszlopokhoz tartozó tulajdonságokat, hogy azokat elhelyezze egy lista szerkezetű raktárban.

```
1 list_store = gtk_list_store_new(3,
2     G_TYPE_BOOLEAN, // Látható-e
3     G_TYPE_STRING,  // Cím
4     G_TYPE_INT       // Oszlop száma
5 );
6
7 columns = gtk_tree_view_get_columns(to_set);
8 for(li=columns, n=0; li != NULL; li = li->next, ++n)
9 {
```

```

10     g_object_get(G_OBJECT(li->data),
11                 "visible", &visible,
12                 "title", &title,
13                 NULL);
14     gtk_list_store_append(list_store, &iter);
15     gtk_list_store_set(list_store, &iter,
16                         0, visible,
17                         1, title,
18                         2, n,
19                         -1);
20 }
21
22 g_list_free(columns);

```

A programrészlet 7. sorában létrehozuk a fa képernyőelem oszlopait tartalmazó listát, amelynek elemeit a 8–20. sorok közt olvasható ciklusban végigjárjuk. A lista elemeit végigjárva a 10–13. sorban lekérdezzük az oszlopok két tulajdonságát, majd a 14–19. sorok közt ezt a két tulajdonságot elhelyezzük a programrészlet 1–5. sorában létrehozott lista szerkezetű raktárban. Ezt a raktárat a későbbiekben megjeleníthetjük egy másik fa képernyőelemben.

A programrészlet 22. sorában a fa képernyőelem oszlopait jelölő listát megsemmisítjük a `g_list_free()` függvény segítségével.

A következő sorok bemutatják hogyan másolhatjuk vissza a felhasználó által módosított tulajdonságokat.

```

1     gtk_tree_model_get_iter_first(tree_model, &iter);
2
3     do {
4         gtk_tree_model_get(tree_model, &iter,
5                             0, &visible,
6                             2, &n,
7                             -1);
8         column = gtk_tree_view_get_column(to_set, n);
9         g_object_set(G_OBJECT(column),
10                     "visible", visible,
11                     NULL);
12     } while(gtk_tree_model_iter_next(tree_model, &iter));

```

Az első sorban a tulajdonságokat tartalmazó raktár elejére állunk, majd a 3–12. sorok közt olvasható ciklus segítségével végigjárjuk annak összes sorát.

A ciklusban először kiolvassuk, hogy a soron következő oszlop láthatóságát hogyan állította be a felhasználó (4–7. sorok), lekérdezzük a beállí-

tandó fa adott oszlopának címét (8. sor), majd a ,egfelelően beállítjuk az oszlop láthatóságát (9–11. sorok).

8.7.4. A fa képernyőelem jelzései

A fa képernyőelemek esetében különösen fontosak a jelzések, hiszen a legtöbb alkalmazás a fa képernyőelemeken végzett műveletek hatására különféle műveleteket kell, hogy elvégezzen. A legfontosabb jelzések a következők és hatásukra meghívható függvények típusa a következő.

"button-press-event" Ezt a jelzést a GTK programkönyvtár akkor küldi, amikor a felhasználó lenyomja az egér valamelyik gombját a fa képernyőelem felett. Valójában a GTK programkönyvtár ezt az eseményt minden képernyőelem esetén kezeli, a fa képernyőelem esetében azonban különleges jelentősége van. A legtöbb alkalmazásban a fa képernyőelem felett lenyomott egérbillentyű hatására egy felbukkanó menüt jelenítünk meg, amelyik éppen arra a sorra vonatkozik, amelyikre a felhasználó kattintott.

A jelzés hatására visszahívott függvény és az esemény feldolgozásához okvetlenül szükséges függvény típusa a következő.

gboolean függvény(GtkWidget *képernyőelem, GdkEventButton *esemény, gpointer adatok); Az egérkattintás hatására visszahívott függvény, amelynek első paramétere a képernyőelemet jelöli a memóriában. A második paraméter szintén mutató, olyan adatszerkezetet jelöl, amelyben az esemény – az egérkattintás – legfontosabb jellemzői vannak felsorolva.

A függvény harmadik paramétere egy mutató, értékét a visszahívott függvény nyilvántartásba vételekor határozhatjuk meg.

A függvény visszatérési értéke igen fontos, meghatározza, hogy az eseményt – az egérkattintást – a GTK programkönyvtárnak fel kell-e dolgoznia. Ha a függvény visszatérési értéke **TRUE** az azt jelzi, hogy a függvény az egérkattintást kezelte, nincs szükség arra, hogy a GTK beépített eseménykezelője az egérkattintást a szokásos módon kezelje. A függvény a **FALSE** visszatérési értékkel jelezheti, hogy a beépített eseménykezelő futtatására szükség van.

gboolean gtk_tree_view_get_path_at_pos(GtkTreeView *fa, gint x, gint y, GtkTreePath **ösvény, GtkTreeViewColumn **oszlop, gint *cella_x, gint *cella_y); A függvény segítségével lekérdezhetjük, hogy a fa képernyőelem adott képpontja a raktár melyik elemét sorát

tartalmazza. Ezt a függvényt általában akkor használjuk, ha arra vagyunk kíváncsiak, hogy az egérmutató alatt a fa melyik sora és oszlopa található.

A függvény paramétereinek és visszatérési értékének jelentése a következő:

fa A fa képernyőelemet jelölő mutató.

x A lekérdezendő koordináta x értéke.

y A lekérdezendő koordináta y értéke.

ösvény Mutató, ami azt a helyet jelzi a memóriában, ahová a függvény az adott ponton megjelenő sor ösvényét tartalmazó memóriaterület címét helyezi el. Ha a függvény a raktár sorát megtalálja és egy ösvény címét helyezi el itt, használat után az ösvényt a `gtk_tree_path_free()` függvénnyel fel kell szabadítanunk.

oszlop Ha ez a paraméter nem `NULL`, akkor olyan memóriaterületet jelöl, ahová a függvény annak a oszlopnak a címét helyezi el, amelyen belül a paraméterként átadott x és y koordináta tartozik. Ezt az adatszerkezetet természetesen nem szerencsés felszabadítani, mert a fa képernyőelem használja.

cella_x, cella_y Ha ezeknek a paramétereknek az értéke nem `NULL`, akkor a függvény ezekre a helyekre elhelyezi az x, y koordináták cellán belüli koordinátarendszerre átszámított értékét. Ezekre az értékekre általában nincsen szükségünk, ezért az utolsó két paraméter értéke általában `NULL`.

visszatérési_érték A függvény visszatérési értéke igaz, ha az adott koordinátákon valóban volt elem és így az ösvényt sikerült megállapítani. A visszatérési érték csak akkor hamis, ha a felhasználó a fa képernyőelemen belül a sorok alatt található üres helyre kattintott.

"row-activated" Ez a jelzés igen fontos, akkor jelentkezik, ha a felhasználó a fa képernyőelem valamelyik sorára dupla kattintással kattint. A jelzést a GTK akkor is küldi, ha a sor aktív oszlopa nem szerkeszthető és a felhasználó a sort a billentyűzettel aktiválja.

A jelzés kezelésére a következő típusú visszahívott függvény használható:

```
void függvény(GtkTreeView *fa, GtkTreePath *path,
               GtkTreeViewColumn *column, gpointer adatok);
```

Ilyen függvényt használhatunk a fa képernyőelem sorának aktiválásához kapcsolódó visszahívott függvényként.

A függvény első paramétere a fa képernyőelemet jelöli a memóriában. A második paraméter az aktivált sort jelölő ösvény, a harmadik paraméter pedig az aktivált oszlopot memóriabeli címe. A visszahívott függvénynek sem az ösvényt, sem pedig az oszlopot nem szabad felszabadítania.

A GTK nem küldi a **"row-activated"** jelzést, ha a felhasználó a fa képernyőelemen belül megjelenített sorok alatti esetlegesen üresen hagyott területre kattint, ezért a visszahívott függvény második és harmadik paramétere mindig érvényes adatszerkezetet jelöl, azaz mindig ismert a sor és az oszlop, ahol az aktiválás történt.

A függvény negyedik paramétere a függvény nyilvántartásba vételekor meghatározott memóriacím, amelyet a programozó kiegészítő adatok átadására használhat.

A következő példa bemutatja hogyan jeleníthetünk meg felbukkanó menüt a fa képernyőelem felett, ha a felhasználó lenyomja az egér jobb egérgombját.

58. példa. A következő függvény egy fa képernyőelemhez tartozó visszahívott függvény, amelyet a GTK programkönyvtár az egér gombjának lenyomásakor – a **"button-press-event"** esemény hatására – hív meg. A függvény megjelenít egy felbukkanó menüt és a menünek átadja annak a sornak az azonosítóját, amelyen a felhasználó az egér gombját lenyomta.

```

1  gboolean
2  on_commissions_treeview_button_press_event(
3      GtkWidget      *widget,
4      GdkEventButton *event,
5      gpointer        user_data)
6  {
7      GtkTreePath      *path;
8      GtkTreeModel      *tree_model;
9      GtkTreeIter       iter;
10     gboolean          found;
11     static GtkWidget *menu_widget = NULL;
12     gint              *id = g_new(gint, 1);
13
14     if (event->type != GDK_BUTTON_PRESS ||
15         event->button != 3)
16         return FALSE;
17
18     found = gtk_tree_view_get_path_at_pos(
19         GTK_TREE_VIEW(widget),

```

```

20         (gint) event->x,
21         (gint) event->y,
22         &path, NULL,
23         NULL, NULL);
24
25     if (found) {
26         tree_model = gtk_tree_view_get_model(
27             GTK_TREE_VIEW(widget));
28         gtk_tree_model_get_iter(tree_model, &iter, path);
29         gtk_tree_model_get(tree_model, &iter, 0, id, -1);
30         gtk_tree_path_free(path);
31     } else {
32         *id = -1;
33     }
34
35     if (menu_widget == NULL)
36         menu_widget = create_tree_view_popup_menu();
37     g_object_set_data(G_OBJECT(menu_widget), "tree_view",
38         widget);
39     g_object_set_data_full(G_OBJECT(menu_widget), "id",
40         id, g_free);
41
42     gtk_menu_popup(GTK_MENU(menu_widget),
43         NULL, NULL, NULL, NULL,
44         event->button, event->time);
45
46     return TRUE;
47 }

```

A függvény a 11. sorban egy statikus mutatót hoz létre, amiben a létrehozott felbukkanó menü címét tárolja. Azért szerencsés statikus mutatót használni, mert a felbukkanó menü nem semmisül meg automatikusan és így a következő függvényhíváskor is felhasználható.

A 12. sorban dinamikusan foglalunk memóriát egy azonosító tárolására. Ezt az azonosítót fogjuk felhasználni arra, hogy a felbukkanó menü menüpontjainak visszahívott függvényei számára jelezzük, hogy a műveletet melyik elemen kell elvégezni. A felbukkanó menüek számára átadott értékeket statikus változóban vagy dinamikusan foglalt memóriaterületen kell tárolnunk, hiszen a függvény végrehajtása után is szükségünk van rájuk.

A függvény 14–16. sorában láthatjuk hogyan ellenőrizhetjük le, hogy a bekövetkezett esemény a jobb oldali nyomógomb lenyomása volt-e. Ha a függvény úgy találja, hogy nem az egér nyomógombjának lenyomása, hanem a nyomógomb felengedése következett be, vagy a nyomógomb sor-

326

száma nem 3, azaz nem a jobb oldali gombról van szó, akkor a 16. sorban a függvény hamis logikai értéket ad vissza. Ezzel jelzi a függvény, hogy az alapértelmezett eseménykezelőnek le kell futnia, hogy az eseményt a szokásos módon kezelje.

Ha a jobb egérgomb lenyomása következett be, akkor ezek után a függvény a 18–23. sorokban a `gtk_tree_view_get_path_at_pos()` függvény segítségével lekérdezi az egérmutató alatt található sorhoz tartozó ösvényt. Ha a mávelet sikeres volt, azaz a felhasználó nem a sorok alatt található üres területre kattintott, akkor a 26–30. sorban lekérdezzük a fa képernyőelemhez tartozó raktárból az adott sor első oszlopában található egész típusú értéket és az ösvény tárolására használt memóriaterületet felszabadítjuk. Ha az egérgomb alatt nincs adatsor, az azonosítót tároló adatterületre `-1`-et helyezünk.

Ezek után, ha a felbukkanó menüt még nem hoztuk létre, a 36. sorban létrehozzuk. A következő lépés a fa képernyőelem és az azonosító átadása a felbukkanó menünek. Ehhez a G programkönyvtár `g_object_set()` és `g_object_set_full()` függvényeit használjuk, amelyekkel a G programkönyvtár által kezelt adatszerkezetekhez rendelhetünk adatokat. Figyeljük meg, hogy a dinamikusan foglalt memóriaterület felszabadítására szolgáló függvényt hogyan adjuk meg a 40. sorban!

Az utolsó lépés a felbukkanó menü megjelenítése a `gtk_menu_popup()` függvénnyel (42–44. sorok), amely után a függvény igaz logikai értéket ad vissza, amellyel jelzi a GTK programkönyvtár számára, hogy az esemény kezelését elvégezte, az alapértelmezett eseménykezelő futtatására nincs szükség.

A következő példa a fa képernyőelemen történő dupla kattintás kezelésére szolgáló visszahívott függvény készítését mutatja be.

59. példa. A következő visszahívott függvény a fa képernyőelem bármely sorának aktiválásakor (dupla kattintás) hívódik meg. A függvény megjelenít egy új ablakot és átadja az ablaknak az aktivált sor azonosítóját.

```

1 void
2 on_this_treeview_row_activated(
3     GtkTreeView      *treeview,
4     GtkTreePath      *path,
5     GtkTreeViewColumn *column,
6     gpointer          user_data)
7 {
8     GtkWidget *window;
9     gint *id = g_new(gint, 1);
10
11     *id = w_tree_view_read_int(treeview, path, 0);
12

```

```

13     window = create_window2();
14     g_object_set_data_full(G_OBJECT(window),
15         "id", id, g_free);
16     gtk_widget_show(window);
17 }

```

A függvény 9. sorában dinamikus memóiafoglalással készítünk elő helyet az aktivált sor szám jellegű azonosítójának tárolására. Mivel a megjelenítendő ablaknak akkor is szüksége lesz erre az azonosítóra, amikor ez a visszahívott függvény már befejeződött, a dinamikus memóiafoglalás szerencsés megoldás.

A 11. sorban a raktárnak a felhasználó által aktivált sorából olvasunk. Az olvasás során egy saját készítésű függvényt használunk, amelyet bárki könnyedén elkészíthet a már bemutatott eszközök felhasználásával.

A 13. sorban létrehozuk az új ablakot a Glade által készített függvény segítségével, a 14–15. sorokban hozzákapcsoljuk az azonosítót, a 16. sorban pedig megjelenítjük a képernyőn.

8.8. Az ikonmező

Az ikonmező a képernyő olyan területe, ahol sorokba és oszlopokba rendezett ikonok és ikonfeliratok jeleníthetők meg. A GTK+ programkönyvtár esetében az ikonmező kezelésére használt típus a `GtkIconView`. A 8.14. ábrán egy ikonmezőt figyelhetünk meg, amely a háttértár könyvtárbejegyzéseit ábrázolja.

Az ikonmező első pillantásra bonyolultnak és nehezen kezelhetőnek tűnik, a programozó szempontjából mégis viszonylag egyszerű eszköz, amelyet néhány egyszerű függvény segítségével kezelhetünk. Az ikonmező számára a megjelenítendő adatokat a `GtkTreeModel` típusú adatszerkezetben adhatjuk meg, de mivel a `GtkListStore` típusú raktár használható `GtkTreeModel` típusként, egyszerű lista típusú raktárat is használhatunk.

Az ikonmező készítéséhez és használatához elengedhetetlenül fontosak a következő függvények:

```

void gtk_icon_view_set_model(GtkIconView *icon_view,
    GtkTreeModel *model);

```



8.14. ábra. Az ikonmező

A függvény segítségével beállíthatjuk, hogy az ikonmező honnan vegye a megjelenítendő ikonok adatait. A függvény első paramétere a beállítandó ikonmezőt, a második paramétere pedig adattárat jelöli a memóriában.

Az ikonmező egyszerű lista adatait képes megjeleníteni, ezért a második paraméterként átadott mutató általában `GtkListStore` típusú elemet jelöl a memóriában.

Fontos megjegyeznünk, hogy ennek a függvénynek a hívása még nem elégséges a működéshez, hiszen azt is meg kell adnunk, hogy az adott lista egyes oszlopai milyen adatokat tartalmaznak, hogy az ikonlista a listán belül hol találja a megjelenítendő képeket és képaláírásokat.

```
void gtk_icon_view_set_text_column(GtkIconView
    *icon_view, gint oszlopszám);
```

A függvény segítségével megadhatjuk, hogy az ikonmezőhöz tartozó lista szerkezetű raktár (`GtkListStore`) melyik oszlopa tartalmazza a képek alatt megjelenítendő szövegeket. A függvény első paramétere az ikonmezőt jelöli a memóriában, a második paramétere pedig megadja, hogy a aláírásként megjelenítendő szövegeket a lista melyik oszlopában helyeztük el. Az oszlopok számozása 0-tól indul.

Ha az ikonmező létrehozása során nem használjuk ezt a függvényt, a megjelenített ikonok alatt nem lesz felirat.

```
void gtk_icon_view_set_pixbuf_column(GtkIconView
    *icon_view, gint oszlopszám);
```

A függvény nagyon hasonló a `gtk_icon_view_set_text_column()` függvényhez, de nem az ikonok alatt megjelenő szövegekre, hanem magukra az ikonokra vonatkozik.

```
void gtk_icon_view_set_item_width(GtkIconView *icon_view,
    gint szélesség);
```

A függvény segítségével megadhatjuk, hogy az ikonmező az ikonok és az aláírások megjelenítéséhez hány képpontnyi szélességű képernyőterületet használjon fel. A függvény első paramétere az ikonmezőt jelöli a memóriában, a második paramétere pedig a képernyőn megjelenő oszlopok képpontban mért szélessége.

Ezt a függvényt tulajdonképpen nem kötelező használni, de igen hasznos lehet, mert ha a GTK+ programkönyvtárra bízunk a szélesség megválasztását, valószínűleg csalódunk.

A következő példa bemutatja az ikonmező egyszerű használatát, amely az eddig bemutatott eszközökkel is lehetséges.

60. példa. A következő programrészlet könyvtárbejegyzéseket jelenít meg egy ikonmezőben. A könyvtárakat mappát, az állományokat dokumentumot formázó ikon jelöli. Ha a felhasználó egy mappát ábrázoló ikonra kattint duplán, az ikonmező tartalma megváltozik, az adott könyvtárban található könyvtárbejegyzéseket jeleníti meg.

```

1  #ifndef PIXMAP_DIR
2  #define PIXMAP_DIR PACKAGE_DATA_DIR "/pixmaps/" \
3      PACKAGE "/"
4  #endif
5  /*
6   * Ez a függvény feltölti a GtkListStore típusú listát
7   * a munkakönyvtárban található könyvtárbejegyzések
8   * adataival.
9   * A lista oszlopai a következők:
10  * 0) A könyvtárbejegyzés neve.
11  * 1) A könyvtárbejegyzés ikonja(könyvtár vagy fájl).
12  * 2) Logikai érték, amely igaz, ha a könyvtár-
13  *     bejegyzés könyvtár.
14  */
15  static void
16  fill_store_with_filenames(GtkListStore *store)
17  {
18      DIR *directory;
19      struct dirent *entry;
20      GdkPixbuf *pixbuf_folder;
21      GdkPixbuf *pixbuf_file;
22      GtkTreeIter iter;
23      /*
24       * Töröljük a lista eredeti tartalmát.
25       */
26      gtk_list_store_clear(store);
27      /*
28       * Létrehozzuk a két használt ikont.
29       */
30      pixbuf_folder = gdk_pixbuf_new_from_file(
31          PIXMAP_DIR "folder.png", NULL);
32      pixbuf_file = gdk_pixbuf_new_from_file(
33          PIXMAP_DIR "file.png", NULL);
34      /*
35       * Megnyitjuk a könyvtárat olvasásra.
36       */
37      directory = opendir(".");
38      if (directory == NULL)

```

330

```

39         return;
40     /*
41      * Olvassuk a könyvtárat és elhelyezzük az elemeket a
42      * listában.
43      */
44     while ((entry = readdir(directory)) != NULL ){
45         /* Beszúrás... */
46         gtk_list_store_append(store, &iter);
47         /* az új elem által hordozott adatok. */
48         gtk_list_store_set(store,
49             &iter,
50             0, entry->d_name,
51             1, entry->d_type == DT_DIR
52                 ? pixbuf_folder : pixbuf_file,
53             2, entry->d_type == DT_DIR,
54             -1);
55     }
56 }
57
58 /*
59  * Ez a függvény egy új GtkListStore listát hoz létre
60  * az újonan készített GtkIconView típusú ikonmező
61  * számára, majd a megfelelő függvény hívásával kitölti
62  * a munkakönyvtár elemeinek neveivel.
63  */
64 void
65 on_iconview_realize(GtkWidget *widget,
66                     gpointer    user_data)
67 {
68     GtkTreeStore *list_store; /*FIXME: típus!*/
69
70     list_store = gtk_list_store_new(3,
71         G_TYPE_STRING,
72         GDK_TYPE_PIXBUF,
73         G_TYPE_BOOLEAN
74     );
75
76     gtk_icon_view_set_model(GTK_ICON_VIEW(widget),
77         GTK_TREE_MODEL(list_store));
78     gtk_icon_view_set_text_column(GTK_ICON_VIEW(widget),
79         0);
80     gtk_icon_view_set_pixbuf_column(GTK_ICON_VIEW(widget),
81         1);
82     gtk_icon_view_set_item_width(GTK_ICON_VIEW(widget),

```

```

83         100);
84
85     fill_store_with_filenames(list_store);
86 }
87
88 /*
89  * Ezt a függvényt akkor hívjuk, ha a felhasználó a
90  * dupla kattintással könyvtárat vált. A függvény
91  * elvégzi a könyvtársváltást, majd a megfelelő
92  * függvénnyel újraolvassa a könyvtárbejegyzések
93  * listáját.
94  */
95 void
96 on_iconview_item_activated(GtkIconView *iconview,
97                             GtkTreePath *path,
98                             gpointer     user_data)
99 {
100     GtkListStore *store;
101     GtkTreeIter iter;
102     gboolean is_dir;
103     gchar *name;
104     /*
105      * Az ikonmezőhöz tartozó lista.
106      */
107     store = GTK_LIST_STORE(
108         gtk_icon_view_get_model(GTK_ICON_VIEW(iconview)));
109     /*
110      * A kijelölt elemhez tartozó bejáró.
111      */
112     gtk_tree_model_get_iter(GTK_TREE_MODEL(store), &iter,
113                             path);
114     gtk_tree_model_get(GTK_TREE_MODEL(store), &iter,
115                         0, &name,
116                         2, &is_dir,
117                         -1);
118     /*
119      * Ha nem könyvtár, nem tudunk könyvtárat váltani.
120      */
121     if (!is_dir) {
122         g_free (name);
123         return;
124     }
125     /*
126      * Könyvtársváltás és újraolvasás.

```


332

```

127     */
128     chdir(name);
129     fill_store_with_filenames(store);
130 }
```

A példaprogram 1–4. sora az alkalmazáshoz tartozó ikonokat tároló könyvtár pontos nevét adja meg. Az itt használt eszközökről bővebben olvashatunk a 346. oldalon kezdődő 9.3.1. szakaszban.

GtkTreeModel *gtk_icon_view_get_model(GtkIconView *ikonmező); E függvény segítségével lekérdezhetjük az ikonmezőhöz tartozó raktár címét.

A függvény paramétere az ikonmezőt jelöli a memóriában, a visszatérési értéke pedig az ikonmezőhöz tartozó raktár címe a memóriában. A visszatérési érték lehet **NULL** is, ha az ikonmezőhöz még nem adtunk raktárat.

gint gtk_icon_view_get_text_column(GtkIconView *ikonmező); A függvény segítségével lekérdezhetjük, hogy az ikonmező a hozzá tartozó raktár melyik oszlopából olvassa a megjelenítendő szöveges értéket. Az oszlop beállítására a már bemutatott **gtk_icon_view_set_text_column()** függvényt használhatjuk

A függvény paramétere az ikonmezőt jelöli a memóriában, visszatérési értéke pedig megadja a szöveges értéket tartalmazó oszlop számát. A visszatérési érték **-1** is lehet, ha a szöveges értéket tartalmazó oszlop számát még nem állítottuk be.

void gtk_icon_view_set_markup_column(GtkIconView *ikonmező, gint oszlop); A függvény segítségével beállíthatjuk, hogy az ikonmező a hozzá tartozó raktár melyik oszlopából olvassa a megjelenítendő szöveges értéket. Ha a szöveges oszlop beállítására ezt a függvényt használjuk, az ikonmező a szöveg megjelenítésekor a Pango jelölőnyelvet is értelmezni fogja a szövegben.

A függvény első paramétere az ikonmezőt jelöli a memóriában, a második paramétere pedig megadja, hogy a hozzá tartozó raktár melyik oszlopa tartalmazza a megjelenítendő adatok Pango jelölőnyelvet is tartalmazó szöveges értékét.

gint gtk_icon_view_get_markup_column(GtkIconView *ikonmező); A függvény segítségével lekérdezhetjük, hogy a Pango jelölőnyelvel készített szöveges érték számára melyik oszlop van beállítva.

A függvény paramétere az ikonmezőt jelöli a memóriában, a visszatérési értéke pedig a jelölőnyelvet használó oszlop száma, ha azt már beállítottuk, vagy **-1**, ha nem.

`gint gtk_icon_view_get_pixbuf_column(GtkIconView *ikonmező);` A függvény segítségével lekérdezhetjük, hogy az ikonmező a hozzá tartozó raktár melyik oszlopából olvassa a megjelenítendő adatok kép jellegű értékét. Ennek az oszlopnak a beállítására a már bemutatott `gtk_icon_view_set_pixbuf_column()` függvényt használhatjuk.

A függvény paramétere az ikonmezőt jelöli a memóriában, a visszatérési értéke pedig megadja az ikonokat tartalmazó oszlop számát. Ha az ikonokat tároló oszlop számát még nem állítottuk be, a visszatérési érték `-1` lesz.

`void gtk_icon_view_set_orientation(GtkIconView *ikonmező, GtkOrientation irány);` A függvény segítségével beállíthatjuk, hogy az ikonmező a szöveges értéket az ikonok alatt, vagy az ikonok mellett jelenítse-e meg.

A függvény első paramétere az ikonmezőt jelöli a memóriában, a második paramétere pedig a megjelenítés módját határozza meg. E második paraméter a következő állandók egyike lehet:

`GTK_ORIENTATION_HORIZONTAL` A szöveges érték az ikonok mellett jelenik meg.

`GTK_ORIENTATION_VERTICAL` A szöveges érték az ikonok alatt jelenik meg. Ez a beállítás az alapértelmezett megjelenítési mód.

`GtkOrientation gtk_icon_view_get_orientation(GtkIconView *ikonmező);` A függvény segítségével lekérdezhetjük a megjelenítés módját.

A függvény paramétere az ikonmezőt jelöli a memóriában, a visszatérési értéke pedig az előző függvéynél bemutatott állandók egyike, amelyek megadják, hogy a címkék az ikonok mellett, vagy alatt jelennek meg.

`void gtk_icon_view_set_spacing(GtkIconView *ikonmező, gint távolság);` A függvény segítségével beállíthatjuk, hogy hány képpontnyi üres hely jelenjen meg az ikonok és a szöveges értékek közt a képernyőn.

A függvény első paramétere az ikonmezőt jelöli a memóriában, a második paramétere pedig megadja az ikonok és a szöveges értékek közt kihagyandó üres hely méretét.

`void gtk_icon_view_set_row_spacing(GtkIconView *ikonmező, gint távolság);` A függvény segítségével beállíthatjuk az ikonmezőben megjelenő sorok közt kihagyandó üres hely méretét képpontban mérve.

334

A függvény első paramétere az ikonmezőt jelöli a memóriában, a második paramétere pedig a sorok közti távolság méretét adja meg.

`void gtk_icon_view_set_column_spacing(GtkIconView *ikonmező, gint távolság);` A függvény segítségével beállíthatjuk, hogy az ikonmező oszlopai közt hány képpontnyi üres terület jelenjen meg.

A függvény első paramétere az ikonmezőt jelöli a memóriában, a második paramétere pedig az egyes oszlopok közt kihagyandó üres hely méretét határozza meg.

`void gtk_icon_view_set_margin(GtkIconView *ikonmező, gint margó);` A függvény segítségével megadhatjuk, hogy az ikonmezőn belül, a széleken mekkora margó jelenjen meg. Az ikonmező szélein a margó minden irányban – alul felül, valamint jobb- és bal oldalon – azonos méretű margó jelenik meg.

A függvény első paramétere az ikonmezőt jelöli a memóriában, második paramétere pedig a margó méretét adja meg.

`void gtk_icon_view_set_reorderable(GtkIconView *ikonmező, gboolean átrendeozhető);` A függvény segítségével megadhatjuk, hogy az ikonmezőben megjelenő ikonok sorrendjét a felhasználó az egérrel átrendeozheti-e.

A függvény első paramétere az ikonmezőt jelöli a memóriában, második paramétere pedig egy logikai érték, ami meghatározza, hogy a felhasználó átrendeozheti-e az ikonok sorrendjét.

Az ikonmezőben a fa képernyőelemhez hasonlóképpen a beállításoktól függően egy vagy több elem lehet kijelölve, sőt a fa képernyőelemhez hasonlóan kezelhetjük a kijelölt elemet vagy elemeket. A legfontosabb függvények a következők.

`void gtk_icon_view_set_selection_mode(GtkIconView *ikonmező, GtkSelectionMode mód);` A függvény segítségével beállíthatjuk, hogy az ikonmezőben egyszerre hány elem lehet kijelölve.

A függvény első paramétere az ikonmezőt jelöli a memóriában, a második paramétere pedig a kijelölés módját határozza meg. Második paraméterként a fa képernyőelemnél már megismert állandókat használhatjuk:

GTK_SELECTION_NONE Az ikonmezőben egyetlen elem sem lehet kijelölve.

GTK_SELECTION_SINGLE Ebben az üzemmódban a felhasználó legfeljebb egy elemet jelölhet ki, de a kijelölést teljesen meg is szüntetheti.

GTK_SELECTION_BROWSE Ebben az üzemmódban a felhasználó pontosan egy elemet jelölhet ki, a kijelölést azonban nem szüntetheti meg csak egy új elem kijelölésével, hogy mindig legyen kijelölt elem. A kijelölési módok közül ez az alapértelmezett viselkedés.

Amikor viszont az ikonmező megjelenik a képernyőn, alapértelmezés szerint nincs egyetlen kijelölt elem sem. A programozónak magának kell arról gondoskodnia, hogy az ikonmező létrehozásakor kijelöljön egy képernyőelemet, ha azt akarja, hogy mindig legyen kijelölt elem.

GTK_SELECTION_MULTIPLE Ebben az üzemmódban egyszerre több elem is ki lehet jelölve.

GtkSelectionMode `gtk_icon_view_get_selection_mode(GtkIconView *ikonmező);` A függvény segítségével lekérdezhetjük az ikonmező kijelölési módját.

A függvény paramétere az ikonmezőt jelöli a memóriában, a visszatérési értéke pedig az előző függvénynél bemutatott állandók egyike, ami megadja az ikonmező kijelölési módját.

void `gtk_icon_view_selected_foreach(GtkIconView *ikonmező, GtkIconViewForeachFunc függvény, gpointer adatok);` A függvény segítségével bejárhatjuk az ikonmezőben kijelölt elemeket és minden kijelölt elemre meghívhatunk egy függvényt.

A függvény első paramétere az ikonmezőt jelöli a memóriában. A második paraméter a kijelölt elemekkel meghívandó függvény címe, a harmadik paraméter pedig a függvénynek átadandó kiegészítő információkat jelöli a memóriában.

void `függvény(GtkIconView *ikonmező, GtkTreePath *ösvény, gpointer adatok);` Ilyen típusú függvényt tudunk meghívni az ikonmező kijelölt elemeire.

A függvény első paramétere az ikonmezőt jelöli a memóriában. A függvény második paramétere egy ösvényt határoz meg. Az ösvény a kijelölt elemek közül a soron következőt jelöli az ikonmezőhöz tartozó raktárban.

A függvény harmadik paramétere a kiegészítő adatokat jelöli a memóriában, amelyeket a `gtk_icon_view_selected_foreach()` függvény utolsó paramétereként megadtunk.

336

gboolean **gtk_icon_view_path_is_selected**(**GtkIconView** *ikonmező, **GtkTreePath** *ösvény); A függvény segítségével lekérdezhethetjük, hogy az ikonmezőhöz tartozó raktár adott sora az adott pillanatban ki van-e jelölve.

A függvény első paramétere az ikonmezőt jelöli a memóriában, a második paramétere pedig az ösvényt, ami a vizsgálandó sort jelöli a raktárban. A függvény visszatérési értéke **TRUE**, ha a sor ki van jelölve, illetve **FALSE**, ha nincs.

GList ***gtk_icon_view_get_selected_items**(**GtkIconView** *ikonmező); E függvény segítségével egy lépésben kérdezhethetjük le az ikonmezőben kijelölt összes elem ösvényét.

A függvény paramétere az ikonmezőt jelöli a memóriában. A visszatérési érték a függvény által létrehozott listát jelöli a memóriában. A lista minden eleme egy mutatót hordoz, ami a raktár egy kijelölt sorát jelölő ösvény mutatója. Használat után a lista elemeiként kapott ösvényeket meg kell semmisítenünk a **gtk_tree_path_free()** függvénnyel és a lista tárolására használt memóriaterületet is fel kell szabadítanunk a **g_list_free()** függvénnyel.

Az ikonmezőben a programunk segítségével az egyes elemek kijelölését megváltoztathatjuk. Erre a következő függvényeket használhatjuk.

void **gtk_icon_view_select_path**(**GtkIconView** *ikonmező, **GtkTreePath** *ösvény); A függvény segítségével az ikonmezőben az ikonmezőhöz tartozó raktár adott sorát kijelölhetjük.

A függvény első paramétere az ikonmezőt, második paramétere pedig a kijelölendő sort azonosító ösvényt jelöli a memóriában.

void **gtk_icon_view_unselect_path**(**GtkIconView** *ikonmező, **GtkTreePath** *ösvény); A függvény segítségével az ikonmezőben az ikonmezőhöz tartozó raktár megadott sorának kijelölését megszüntethetjük.

A függvény első paramétere az ikonmezőt, a második paramétere pedig az ikonmezőhöz tartozó raktár egy sorát kijelölő ösvényt jelöli a memóriában.

void **gtk_icon_view_select_all**(**GtkIconView** *ikonmező); A függvény segítségével az ikonmező összes elemét egy lépésben kijelölhetjük.

A függvény paramétere az ikonmezőt jelöli a memóriában, amiben az összes elemet ki akarjuk jelölni.

```
void gtk_icon_view_unselect_all(GtkIconView *ikonmező);
```

A függvény segítségével az ikonmezőben található összes elem kijelölését megszüntethetjük.

A függvény paramétere az ikonmezőt jelöli a memóriában.

A fa képernyőelemhez hasonlóan az ikonmezőben is értelmezhetjük a kurzor fogalmát. A kurzor az ikonmező egy elemét azonosítja, az aktív elemet, amelyet a felhasználó az Enter billentyű lenyomásával szerkeszthet, ha az ikonmező szerkeszthető. (A szerkeszthető ikonmező készítése a 8.8.1. szakaszban visszatérünk.) Az ikonmező aktív elemét kijelölő kurzort a következő függvényekkel kezelhetjük:

```
void gtk_icon_view_set_cursor(GtkIconView *ikonmező,
    GtkTreePath *ösvény, GtkCellRenderer *cellarajzoló,
    gboolean szerkeszt);
```

E függvény segítségével beállíthatjuk az ikonmezőhöz tartozó kurzort.

A függvény első paramétere az ikonmezőt jelöli a memóriában. A második paraméter az ikonmezőben található valamelyik cellaranjzólót jelöli vagy `NULL`. Ha magunk hoztuk létre az ikonmező cellaranjzolóit, hogy szerkeszthetővé tegyük őket, akkor általában megadjuk a cellarajzoló címét, ha pedig az eddig bemutatott egyszerűbb eszközökkel kezeljük az ikonmezőt, általában egyszerűen `NULL` értéket adunk meg a második paraméter helyén.

A függvény harmadik paraméterével meghatározhatjuk, hogy a kurzor áthelyezése után a cellarajzolóban automatikusan megkezdődjön-e a szerkesztés. Ha ennek a paraméternek az értéke igaz, a cellarajzoló automatikusan szerkesztő üzemmódba kapcsol.

```
gboolean gtk_icon_view_get_cursor(GtkIconView *ikonmező,
    GtkTreePath **ösvény, GtkCellRenderer **cellarajzoló);
```

A függvény segítségével lekérdezhetjük az ikonmező kurzorának helyét.

A függvény első paramétere az ikonmezőt jelöli a memóriában. A második paramétere azt a memóriaterületet jelöli, ahová a függvény a kurzor helyét jelölő ösvény címét elhelyezi. E paraméter értéke lehet `NULL` is, ekkor a függvény nem adja vissza a kurzor helyét jelölő ösvényt.

A függvény harmadik paramétere azt a memóriaterületet jelöli, ahová a függvény az aktív cellarajzoló címét elhelyezi. E paraméter értéke is lehet `NULL`.

A függvény visszatérési értéke jelzi, hogy a kurzor érvényes-e, azaz volt-e aktív elem az ikonmezőben.

8.8.1. Az ikonmező cellarajzolói

Az ikonmező kezelésére használt `GtkIconView` típus megvalósítja a `GtkCellLayout` interfészt és így cellaelrendezésként használva cellarajzolókat fogadására is alkalmas. Ha az ikonmezőben található adatok szerkesztését is lehetővé akarjuk tenni a felhasználó számára, a legegyszerűbb, ha a cellarajzolókat magunk készítjük el és magunk helyezzük el az ikonmezőben.

8.8.2. Az ikonmező jelzései

Az ikonmező használatában a fa képernyőelemhez hasonlóan fontos szerepet játszanak a jelzések. Az ikonmező által küldött legfontosabb jelzéseket a következő lista mutatja be.

"button-press-event" A GTK programkönyvtár akkor küldi ezt a jelzést, ha a felhasználó lenyomja (vagy felengedi) az egér valamelyik nyomógombját az ikonmező területén belül. Ezt a jelzést minden képernyőelemnél használhatjuk, de az ikonmező esetében különleges jelentősége van, mert az ikonmezőn általában olyan felbukkanó menüt jelenítünk meg, ami arra az elemre vonatkozik, amelyen az egér gombját a felhasználó lenyomta.

Az esemény kezelésére használt függvények közül a legfontosabbak a következők.

`gboolean` `függvény(GtkWidget *képernyőelem, GdkEventButton *esemény, gpointer adatok);` Ilyen függvényt kell visszahívott függvényként az eseményhez rendelnünk. A függvény működését, a paraméterek és a visszatérési érték jelentését a 322. oldalon már bemutattuk.

`GtkTreePath *gtk_icon_view_get_path_at_pos(GtkIconView *ikonmező, gint x, gint y);` Ennek a függvénynek a segítségével megállapíthatjuk, hogy a felhasználó az ikonmező melyik elemére kattintott.

A függvény első paramétere az ikonmezőt jelöli a memóriában, második és harmadik paramétere pedig megadja, hogy az ikonmező melyik pontját akarjuk lekérdezni.

A függvény visszatérési értéke egy ösvényt jelöl a memóriában, amely a megadott ponthoz tartozó elemet jelöli az ikonmezőhöz tartozó raktárban. Ha a megadott koordinátákon nem található elem, a függvény `NULL` értéket ad vissza.

`gboolean` `gtk_icon_view_get_item_at_pos(GtkIconView *ikonmező, gint x, gint y, GtkTreePath **ösvény,`

`GtkCellRenderer **cellarajzoló);` Ennek a függvénynek a segítségével szintén lekérdezhetjük az ikonmező adott koordinátájához tartozó elemet, ez a függvény azonban több információt is képes szolgáltatni.

A függvény első paramétere az ikonmezőt jelöli a memóriában, második és harmadik paramétere pedig megadja a lekérdezendő koordinátákat.

A függvény negyedik paramétere azt a memóriaterületet jelöli, ahová a függvény a keresett elemet jelölő ösvény címét elhelyezi. Ha erre az információra nincs szükségünk a negyedik paraméter helyén `NULL` értéket is megadhatunk.

A függvény ötödik paramétere azt a memóriaterületet adja meg, ahová a függvény a képernyő megadott helyén megjelenő cellarajzoló címét elhelyezi. Az ötödik paraméter helyén is megadható `NULL` érték, ha nincs szükségünk a cellarajzoló címére.

A függvény visszatérési értéke megadja, hogy az ikonmező megadott helyén volt-e cellarajzoló.

"item-activated" Ezt az üzenetet akkor küldi az ikonmező, ha a felhasználó valamelyik ikont dupla kattintással kiválasztja. A legtöbb ikonmező esetében ezt az üzenetet is fogadnunk kell.

A jelzéssel kapcsolatban használható függvények a következők.

`void függvény(GtkIconView *ikonmező, GtkTreePath *ösvény, gpointer adatok);` Ilyen típusú függvényt kell készítenünk az esemény kezelésére.

A függvény első paramétere az ikonmezőt, a második paramétere az aktivált elemet jelző ösvény helyét jelzi a memóriában. A függvény harmadik paramétere a kiegészítő adatok címét adja meg. Ezt a mutatót a visszahívott függvény nyilvántartásba vételekor adhatjuk meg.

`void gtk_icon_view_item_activated(GtkIconView *ikonmező, GtkTreePath *ösvény);` Ennek a függvénynek a segítségével a program aktiválhatja az ikonmező egy elemét éppen úgy, mintha a felhasználó aktiválta volna azt. A függvény első paramétere az ikonmezőt, második paramétere pedig az aktiválandó elemet kijelölő ösvényt jelölöli a memóriában.

340

9. fejezet

Kiegészítő eszközök

A következő oldalakon az alkalmazásfejlesztés néhány fontosabb lépését és eszközét ismerhetjük meg részletesebben. Ezeknek az ismereteknek a birtokában könnyebbé, gördülékenyebbé tehetjük a munkánkat az elkészített alkalmazás minőségét pedig javíthatjuk.

9.1. A forrásprogram terjesztése

A forrásprogramot mindig a szabályos módon elkészített tömörített állományban kell publikálnunk, hogy a program fordítás előtti beállítása és fordítása problémamentes legyen. Soha ne terjeszzük az általunk készített alkalmazás forrásprogramját olyan tömörített állományban, amelyet a `tar` program indításával „kézzel” állítottunk elő!

A forrásprogram terjesztésére alkalmas állomány előállítására a alkalmazás forrásának könyvtárában található `Makefile` állomány használható. A terjesztési állomány létrehozására használatos parancsokat tehát abban a könyvtárban kell kiadnunk, amelyikben a `configure.in` és a `autogen.sh` állomány is található.

Mielőtt a forrásprogram terjesztésére alkalmas állomány előállítanánk, érdemes futtatnunk a `make distcheck` parancsot. A parancs hatására összetett folyamat kezdődik, amelynek során kiderül hogy a létrehozandó terjesztési állomány használható lesz-e.

Az ellenőrzés után a `make dist` parancs hatására a terjesztési állomány automatikusan létrejön.

9.2. A programváltozatok követése

Az alkalmazás fejlesztése során az egyes változatokat számmal (*version number*, változat száma) látjuk el. A Glade kezdetben a **0.1** változatszámmal látja el a programot és ezt a változatszámot el is helyezi a `configure.in` állományban.

Ha növelni akarjuk a változatszámot, egyszerűen csak át kell írunk azt a `configure.in` állományban, majd le kell futtatnunk az `autogen.h` héjprogramot a `./autogen.sh` parancs kiadásával. Ezzel a program változatszáma megváltozik a következőkben létrehozott terjesztési állomány már az új változatszámot tartalmazza.

9.2.1. Különbségi állományok

Az alkalmazások fejlesztését ma már a legtöbb esetben nem egy programozó, hanem programozók egész csoportja végzi, ezért fontos, hogy programozók közti kapcsolattartás minél zökkenőmentesebb legyen. Az egyes programfejlesztők által végzett részfeladatok szétosztásának, a programmódosítások összegyűjtésének és nyilvántartásának segítésére sok fejlett program készült, a legtöbbjük azonban túlságosan bonyolultak ahhoz, hogy néhány szóban bemutassuk. Néhány szóban bemutatjuk viszont a `diff` és `patch` UNIX segédprogramokat, amelyek olyan egyszerűen használhatók a program változásainak figyelemmel követésére, hogy néhány perc alatt elsajátíthatjuk a használatukat.

A `diff` program segítségével két állomány vagy könyvtár különbségi állományát állíthatjuk elő. A különbségi állományban egy helyen olvashatjuk a programozó által végzett különféle változtatásokat, sőt azokat „érvényesíthetjük” azaz az eredeti programváltozatba a változtatásokat bevezethetjük, hogy az új programváltozatot előállítsuk. A szabad szoftvereket fejlesztő közösségben bevett szokássá vált a különbségi állományok elektronikus levélben való továbbítása, ha szabad szoftverek fejlesztésére adjuk a fejünket szinte biztos, hogy a megfelelő formájú különbségi állományt várja tőlünk a fejlesztést koordináló programozó.

A `diff` programnak paraméterként a két összehasonlítandó állományt vagy könyvtárat kell átadnunk, először mindig az eredetit, majd a módosítottat. Ha a változtatás kizárólag egy állományt érint, elegendő annak a különbségi állományát megadni, ha azonban több állományt is módosítottunk, akkor a teljes könyvtárszerkezetet össze kell hasonlítanunk. A különbségi állomány elkészítésekor ismernünk kell a `diff` program következő kapcsolóit:

- u Ennek a kapcsolónak a hatására a `diff` egységesített különbségi állomány formátumot (*unified diff format*) használ, amit a legtöbb programozó megszokott és könnyen tud olvasni.

Fontos, hogy a különbségi állományt olyan formátumban küldjük a többi fejlesztőnek, ami számukra megszokott, ellenkező esetben valószínűleg egyszerűen figyelmen kívül hagyják a levelünket, esetleg az újraküldést kérik. Érdemes tehát ellenőriznünk az alkalmazás dokumentációjában a megkövetelt különbségi állomány formátumát. Ha a dokumentáció nem tér ki a különbségi állomány formátumára, mindenképpen használjuk a `-u` kapcsolót.

- p E kapcsoló hatására a `diff` a különbségi állományban minden változtatás mellett feltünteti azt is, hogy az melyik C függvényben található. Ezt a kapcsolót szintén érdemes használnunk, hiszen nagymértékben megkönnyíti a változtatások nyomon követését.
- r Ennek a kapcsolónak a hatására a `diff` a változtatások keresését minden alkönyvtárban elvégzi. Ha több állományt is módosítottunk és a különbségi állományt két könyvtár összehasonlításával készítjük el, ezt a kapcsolót mindenképpen használnunk kell.
- N Ennek a kapcsolónak a hatására a nem létező állományokat a `diff` úgy kezeli, mintha léteznének, de egyetlen sort sem tartalmaznának. E kapcsolónak nyilvánvalóan akkor van szerepe, ha a program forrásában nem csak a meglévő állományokat módosítottuk, hanem új állományokat is létrehoztunk. Ekkor mindenképpen meg kell adnunk a `-N` kapcsolót, de mivel akkor sem okoz problémát, ha nem hoztunk létre új állományt, általában mindig használjuk.
- X E kapcsoló után megadhatjuk annak az állománynak a nevét, ami a különbségi állomány elkészítésekor figyelmen kívül hagyandó állományok neveit adja meg. Ez igen fontos kapcsoló, hiszen a legtöbb programcsomag rengeteg olyan állományt tartalmaz, amelyek a fordításakor automatikusan jönnek létre és ezért a különbségük nem lényeges.

Ha egyszerűen összehasonlítjuk az eredeti és a módosított program forrásprogramját tartalmazó könyvtárat általában több ezer, vagy akár több tízezer sornyi különbségi állományt kapunk. Az ilyen különbségi állomány publikálása természetesen halálos vétek volna, ezért a legtöbb esetben szükségünk lesz egy, a kihagyandó állományokat felsoroló állományra.

Ha a fejlesztők nem biztosítottak ilyen állományt, mi is könnyedén elkészíthetjük. Egyszerűen át kell olvasnunk a különbségi állományt, ki kell válogatnunk azoknak az állományoknak a nevét, amelyeket ki akarunk hagyni és fel kell sorolnunk a nevüket az állományban.

- B Ennek a kapcsolónak a hatására a program figyelmen kívül hagyja azokat a változtatásokat, amelyek csak üres sorok beszúrására és

törlésére vonatkoznak. Nem kell mindig használnunk ezt a kapcsolót, de érdemes tudnunk róla, hogy van ilyen lehetőségünk, mer néha szükségünk lehet rá..

- b Ennek a függvénynek a hatására a `diff` figyelmen kívül hagyja azokat a módosításokat, amelyek csak a szóközők és tabulátor karakterek számában hoztak változást. Szintén nem szükséges mindig használnunk a `-b` kapcsolót, de érdemes tudnunk róla.

A következő példa bemutatja hogyan használhatjuk a `diff` programot egy programcsomag módosításainak kikeresésére.

61. példa. A következő parancs a `diff` program segítségével összehasonlítja az eredeti állományokat tartalmazó `ak-0.1` könyvtárat a módosított állományok `ak-0.1-hacked` nevű könyvtárával, miközben kihagyja a `dontdiff` állományban felsorolt nevű állományokat.

```
diff -uprbN -X ak-0.1/dontdiff ak-0.1 ak-0.1-hacked/ >ak.diff
```

A `dontdiff` állomány tartalma a következő volt:

1	<code>*.tar.gz</code>
2	<code>*.o</code>
3	<code>*.log</code>
4	<code>*.swp</code>
5	<code>ak</code>

Amint láthatjuk a figyelmen kívül hagyandó állományok felsorolásakor használhatjuk az állománynév helyettesítőkaraktereket, ami nyilvánvalóan nagyon megkönnyíti a munkánkat.

A különbségi állomány érvényesítésére a `patch` programot használhatjuk. A `patch` alapértelmezés szerint a szabványos bemenetről olvassa be a különbségi állományt, amelyben megtalálja, hogy az egyes állományokon milyen módosításokat kell elvégeznie. A program ezek után egyenként ellenőrzi, hogy a változtatások környezete megegyezik-e az eredeti állományban található sorokkal és ha igen, akkor végrehajtja a változtatásokat. Ha a `patch` nem tudja érvényesíteni a változtatásokat, mert azok elvégzése óta az eredeti állomány adott része megváltozott, akkor a nem érvényesített változtatásokat `.rej` nevű állományokba menti.

A változtatások érvényesítése során általában szükségünk van a `diff` következő kapcsolójára:

- p A kapcsoló után egy számot kell megadnunk, ami meghatározza, hogy az eredeti állományok nevéből a `patch` hány könyvtárnevet távolítson el.

Erre a kapcsolóra akkor van szükségünk, ha a különbségi állományt nem abban a könyvtárban akarjuk érvényesíteni, amelyikben készítettük. A legtöbb esetben szükség van ennek a kapcsolónak a használatára.

A következő példa bemutatja hogyan érvényesíthetjük a különbségi állományt a `patch` program segítségével.

62. példa. Az előző példa alapján legyen most az `ak.tmp` különbségi állomány az `ak-0.1` könyvtárban. Próbáljuk meg érvényesíteni a különbségi állományt a `patch` program segítségével (a rövidség kedvéért a kimeneten megjelenő sorok közül néhányat eltávolítottunk):

```
$ patch <ak.diff
can't find file to patch at input line 4
Perhaps you should have used the -p or --strip option?
The text leading up to this was:
-----
|diff -uprbN -X ak-0.1/dontdiff ak-0.7/src/automaton.c ak-0.1-
|hacked/src/automaton.c
|--- ak-0.1/src/automaton.c 2006-11-27 10:19:42.00000000 00 +01
|00
|+++ ak-0.1-hacked/src/automaton.c 2006-11-27 10:54:10.00000000
|00 +0100
...
-----
File to patch:  + 
```

Amint láthatjuk a program nem találta meg a módosítandó állományt, ezért a  +  billentyűkombinációval megszakítottuk a futását.

A képernyőről azt is leolvashatjuk, hogy a `patch` azért nem találta meg a keresett állományt, mert a `ak-0.1-hacked/src` könyvtárban kereste, pedig az a `src/` könyvtárban van. Nyilvánvaló, hogy a módosítandó állományok neveinek elejéről el kell távolítanunk egy könyvtárnevet a `-p` kapcsoló segítségével:

```
$ patch -p 1 <tmp.diff
patching file src/automaton.c
$
```

Amint látható így már semmi sem akadályozta a különbségi állomány érvényesítését.

9.3. Új forrás-állományok

Ha az alkalmazás létrehozásakor úgy döntünk, hogy új forrássállományokkal bővítjük azt, az új állományokat létre kell hoznunk, majd nyilvántartásba kell vennünk és a fordítást vezérlő állományokat is frissítenünk kell.

Az új forrásállományokat szerencsés ugyanabban a könyvtárban létrehozni, amelyikben a már meglévő forrásprogramokat is tároljuk. Ez a könyvtár – amint azt már láttuk – alapértelmezett esetben az alkalmazás forráskönyvtárának `src/` alkönyvtára.

Ha létrehoztuk az új forrásállományokat, azok neveit az adott könyvtárban – azaz az `src/` alkönyvtárban – található `Makefile.am` állományba be kell írunk. Ez nem okozhat komoly problémát, hiszen ebben az állományban már megtalálhatók a Glade által létrehozott állományok nevei, így egyszerűen csak bővítenünk kell a listát.

Utolsó lépésként frissítenünk kell a fordítást vezérlő állományokat az `autogen.sh` héjprogram futtatásával.

Ezzel az új forrásállománnyal kapcsolatos összes feladatunkat elvégeztük, a következő fordításkor már az új forrásállományok is lefordulnak és a következő terjesztési állományból sem fognak hiányozni.

9.3.1. Ikonok elhelyezése a programcsomagban

Amint azt már láttuk a Glade szerkesztőprogrammal ikonokat rendelhetünk az egyes képernyőelemekhez. A munka közben a számítógépre telepített ikonok közül válogathatunk.

Sokszor előfordul azonban, hogy a rendszerre telepített ikonok közül egyik sem fejezi ki pontosan az adott eszköz szerepét, ezért saját ikont szeretnénk felmásolni a számítógépre az alkalmazásunk telepítésekor. A Glade ebben is segít!

A Glade *Projekt* menüjének *Beállítások* menüpontjának segítségével a *képek könyvtára* mezőben beállíthatjuk, hogy a programunk melyik könyvtárában szeretnénk elhelyezni az ikonokat. Itt alapértelmezés szerint a `pixmap`s könyvtárnév szerepel, ami tökéletesen meg is felel az igényeinknek.

Ha tehát a programunk forráskönyvtárában létrehozzuk a `pixmap`s alkönyvtárat és ide másoljuk az ikonokat, amelyeket az alkalmazáshoz készítettünk, az ikonok felmásolásódnak a megfelelő könyvtárba a telepítéskor. Ha megfigyeljük a telepítéskor megjelenő üzeneteket, könnyen nyomonkövethetjük ezt a műveletet.

63. példa. Keressük ki a telepítéskor megjelenő üzenetek közül azokat, amelyek megmutatják mi történik a `pixmap`s könyvtárban található állományokkal (a példában a sorok nagy részét eltávolítottuk, csak a lényeges utasításokat hagytuk meg)!

```
$ make install
...
if test -d ./pixmap; then \
  /home/pipas/prg/install-sh -d /usr/share/pixmap/prg; \
  for pixmap in ./pixmap/*; do \
```

```

        if test -f $pixmap; then \
            /usr/bin/install -c -m 644 $pixmap /usr/share/pixmaps/prg; \
        fi \
    done \
fi
...
$

```

Figyeljük meg, hogy a telepítést végző program megvizsgálja, hogy létezik-e a `pixmaps` könyvtár, és ha igen, annak tartalmát egy újonnan létrehozott könyvtárba másolja!

Az ikonok felmásolása tehát egyszerű művelet, a Glade által létrehozott állományokban található utasítások elvégzik a munkát. A nehezebb feladat az ikonok megkeresése a program futtatásakor, de ebben is segít a Glade.

Az alkalmazás forrását alkotó állományok létrehozásakor a forráskönyvtárban létrejön egy `config.h` állomány, amely a program legfontosabb tulajdonságait tartalmazza. Itt többek közt megtalálhatjuk a következő makrók létrehozását:

PACKAGE A programcsomag neve.

VERSION A programcsomag változatának száma.

A programcsomag fordítása során a `Makefile` a C fordítónak a `-D` kapcsolóval utasítást ad arra, hogy bizonyos makrókat hozzon létre, mielőtt a fordítást elkezd. A létrehozott makrók közül a legfontosabbak a következők:

HAVE_CONFIG_H Ha a makró létezik, a telepítés előtti beállítás létrehozta a `config.h` állományt.

PACKAGE_DATA_DIR A makró szöveges értéke megadja a programunkhoz tartozó adatkönyvtárat, azt a könyvtárat, ahova az adatállományok kerülnek a telepítés során.

Az ikonállományok megkeresése ezek után egyszerűen elvégezhető. Amikor be akarunk tölteni egy képállományt, az itt bemutatott makrók segítségével a fordításkor beállított könyvtárra kell hivatkoznunk. Ezt mutatja be a következő példa.

64. példa. A következő sorok a 329. oldalon található 60. példaprogram részletei, bemutatják hogyan deríthetjük ki, hogy a telepítéskor hova kerültek a képállományok, amelyek a programunkhoz tartoznak.

```

1  #ifndef PIXMAP_DIR
2  #define PIXMAP_DIR PACKAGE_DATA_DIR "/pixmaps/" \

```


348

```

3          PACKAGE "/"
4 #endif
5
6     ...
7
8     pixbuf_folder = gdk_pixbuf_new_from_file(
9         PIXMAP_DIR "folder.png", NULL);

```

A programrészlet 1–4. sorában létrehozunk egy makrót, amelynek az értéke szöveges, megadja a képállományok tárolására használt könyvtár nevét. Figyeljük meg, hogy a makró kihasználja, hogy a C programozási nyelv újabb változatai lehetővé teszik a fordításkor is ismert szöveges állandók egymáshoz fűzését (konkatenációját) az egymás után írás segítségével!

A makrót a programrészlet 9. sorában használjuk. Itt is az egymáshoz fűzés módszerét használjuk, amely egyesíti a könyvtárnevet az állománynévvel.

9.4. Többsz nyelvű programok készítése

A magyar anyanyelvű felhasználóknak nagyon fontos lehet, hogy a felhasználói felület magyarul jelenjen meg, hiszen e nélkül sokan nem is tudják használni a programunkat. A Glade használata közben a magyar nyelvű felhasználói felület elkészítésére két módon is lehetőségünk van.

Az egyszerűbb módszert követve a feliratokat, szövegeket magyar írjuk a felhasználói felületbe, így az elkészült program kizárólag magyar üzeneteket lesz képes használni. A könyv legtöbb példája ezzel a módszerrel készült, hogy a könyvbe ábraként beillesztett képernyőképek magyar feliratokkal jelenjenek meg már a felhasználói felület szerkesztése közben is. Valójában ez a módszer igen szerencsétlen módon csak a magyarul tudó felhasználók számára teszi elérhetővé az elkészült programot, ezért nem igazán javasolható a használata.

A magyar nyelvű felhasználói felület létrehozására tökéletes megoldást a többsz nyelvű felhasználói felület létrehozása (*internationalization, i18n*) a GNU programok általában a `gettext()` eszközcsaládot használják. Erre az eszköztárra építhetjük Glade segítségével készített programokat is.

A módszer lényege, hogy a felhasználói felületet angol nyelvű üzenetekkel látjuk el, az üzeneteket kigyűjtjük a forrásállományokból és lefordítjuk magyarra, majd a telepítés során a fordítást is telepítjük a számítógépre. A felhasználó a `LANG` vagy `LC_ALL` környezeti változó beállításával írhatja elő, hogy milyen nyelven kívánja a felhasználói felületet megjeleníteni.

65. példa. Elkészítettük a *cyclops* nevű programot, felkészítettük az angol és a magyar nyelv támogatására. Szeretnénk úgy elindítani, hogy magyarul jelenjen meg a felhasználói felület. Használjuk a következő parancsot:

```
$ LANG=hu_HU.UTF-8 cyclops
$
```

A parancs futtatja a programot a *LANG* környezeti változó megfelelő beállításával.

Fontos, hogy ha többnyelvű támogatással készítjük el a felhasználói felületet, a Glade használata közben ragaszkodjunk az angol nyelvű szövegek begépeléséhez. Ha magyar feliratokkal látjuk el a felhasználói felületet a program fordításával és telepítésével problémáink lesznek.

A programok több nyelvű felhasználói felülettel való ellátását több lépésben, példákon keresztül mutatjuk be.

Az első lépés a Glade megfelelő beállítása. Ehhez meg kell nyitnunk a projektet, majd ki kell választanunk a *projekt* menü *beállítások* menüpontját. Az itt megjelenő párbeszédablak felső részén ki kell választanunk a *C beállítások* fület, hogy bekapcsolhassuk a *gettext támogatás* címkével ellátott jelölőnégyzetet. Ez a jelölőnégyzet bekapcsolja a *gettext* rendszert, amelyet a legtöbb szabad szoftver használ a többnyelvű felhasználói felületek előállítására. Ha e beállítást elvégeztük, a projektet mentenünk kell és az állományokat létre kell hoznunk. Ha azonban a többnyelvű támogatást az után hozzuk létre, hogy a projektet már létrehoztuk, a Glade nem írja felül a *configure.in* állományt az új beállításokkal. Ekkor például törölhetjük az állományt és újra létrehozhatjuk a projektállományokat, hogy a frissítés megtörténjen.

Ha a *gettext* támogatást bekapcsoltuk a Glade segítségével szerkesztett felhasználói felület elemeit angol nyelven kell megírunk, a fordítást később, más eszközökkel fogjuk elvégezni.

A következő lépés a magyar nyelv támogatásának bekapcsolása lesz. A támogatott nyelvek – a tulajdonképpeni fordítások – listáját a *configure.in* állományban a *ALL_LINGUAS* változóban találjuk, itt kell elhelyeznünk a magyar nyelv szabványos jelölését, ami a *hu* rövidítés. Ezt a lépést mutatja be a következő példa.

66. példa. Keressük ki a támogatott nyelvek listáját és helyezzük el a magyar nyelvet benne! A Glade beállításával kértük a *gettext* támogatás bekapcsolását, a projektet mentettük és az állományokat létrehoztuk. Vizsgáljuk meg a *configure.in* állományt.

1	<i>dnl Add the languages which your application supports</i>
2	<i>dnl here.</i>

350

```
3 ALL_LINGUAS=""
4 AM_GLIB_GNU_GETTEXT
```

```
1 dnl Add the languages which your application supports
2 dnl here.
3 ALL_LINGUAS="hu"
4 AM_GLIB_GNU_GETTEXT
```

Irodalomjegyzék

- [1] Pere László. *Linux: felhasználói ismeretek II.* Kiskapu, Budapest, 2002.
- [2] Tim-Philipp Müller. Gtk+ 2.0 tree view tutorial.

Tárgymutató

' ', 151, 177
'/', 76
'\\', 76
(), 177
(*GFunc)(), 106
(*GOptionArgFunc)(), 119
(*GtkTreeModelForeachFunc)(),
262
*, 171, 177
+, 177
-, 151, 177
., 151
.., 171
..., 185, 189
./autogen.sh, 13, 14, 342
./configure, 14
/, 150, 151, 170, 171, 177
<, 151
< >, 150
<? ?>, 150
<paragraph></paragraph>, 151
=, 151
>, 151
[], 171, 172
#ifdef, 75
#ifndef, 75
#include, 129
ösvény, 262, 323
érték1, 232
_, 151
_PipObjectPrivate, 147
_private, 160
0.1, 342
1.0, 150
ablak, 185, 190
ABS(a), 76
AC_SUBST(), 152
activate, 44
adat, 114
adat, 112, 224, 262
adatifelszabadító, 112
ALL_LINGUAS, 349
arg, 118, 119
arg_data, 119
arg_description, 120
argc, 120, 128
argv, 120
autoconf, 14, 21, 153
autogen.h, 342
autogen.sh, 153, 346
automake, 14, 21, 153
backspace, 44
balra, 227
be, 51
bejáró, 212, 224, 225, 227, 262
blue, 195
BonoboDock, 206
BonoboDockItem, 206, 207
boolval, 176
button_press_event, 58, 59
címsor, 188
cella_x, 323
cella_y, 323
cetlinév, 232
changed, 45
changed, 47, 248
char, 74

children, 155
 children, 160
 children, 110, 162
 CLAMP(x, alacsony, magas), 76
 class, 139
 clicked, 18, 40, 204
 colorsel, 195, 196
 configure, 152, 153
 const void *, 74
 const xmlChar *name, 164
 content, 162
 create_window1(), 180
 create_window2(), 180
 create_xxx(), 57, 180

 data, 50
 description, 120
 dev, 23
 devhelp, 9, 36
 dictionary, 50
 diff, 342–344
 dispose(), 140, 141
 disposed, 134, 141
 doc, 162
 double, 75

 edited, 276
 első_gomb, 188
 első_válasz, 189
 encoding, 150
 enter, 40
 enum, 143
 exit(), 20

 függvény(), 291, 322, 323, 335, 338, 339
 függvény, 113
 függvény, 112, 224
 függvénynév(), 291
 fa, 113
 fa, 323
 FALSE, 42, 51, 58, 70, 73, 76, 93, 101, 114, 121, 124, 189, 205, 218, 220, 221, 227, 230, 231, 237, 244, 254–258, 262, 283, 288, 302, 322, 336
 fflush(), 77
 finalize(), 140, 141
 flags, 118
 float, 75
 floatval, 176
 formázószöveg, 185
 fprintf(), 83, 183
 free(), 80, 157

 g_assert(), 77
 g_assert(kifejezés), 76
 g_assert_not_reached(), 77
 g_build_filename(), 115
 g_build_path(), 115
 g_critical(formátumszöveg, ...), 78
 G_DIR_SEPARATOR, 76
 G_DIR_SEPARATOR_S, 76, 115
 G_DISABLE_ASSERT, 77
 g_error(), 127
 g_find_program_in_path(), 115
 g_fprintf(), 83
 g_free(), 80, 81, 83, 190, 291
 g_getenv(), 116, 117
 G_IS_DIR_SEPARATOR(c), 76
 g_list_append(), 102, 109
 g_list_concat(), 105
 g_list_delete_link(), 104
 g_list_find(), 107
 g_list_find_custom(), 107
 g_list_first(), 106
 g_list_foreach(), 105, 106
 g_list_free(), 104, 313, 319, 321, 336
 g_list_index(), 107
 g_list_insert(), 103
 g_list_insert_before(), 103
 g_list_insert_sorted(), 103, 105, 107
 g_list_last(), 106
 g_list_length(), 105

354

[g_list_next\(\)](#), 106
[g_list_nth\(\)](#), 106
[g_list_nth_data\(\)](#), 106
[g_list_position\(\)](#), 107
[g_list_prepend\(\)](#), 102
[g_list_previous\(\)](#), 106
[g_list_remove\(\)](#), 104
[g_list_remove_all\(\)](#), 104
[g_list_reverse\(\)](#), 105
[g_list_sort\(\)](#), 105
[g_malloc\(\)](#), 79–81
[g_malloc0\(\)](#), 80
[G_MAXDOUBLE](#), 75
[G_MAXFLOAT](#), 75
[G_MAXINT](#), 75
[G_MAXINT16](#), 75
[G_MAXINT32](#), 75
[G_MAXINT64](#), 75
[G_MAXINT8](#), 75
[G_MAXLONG](#), 75
[G_MAXSHORT](#), 75
[G_MAXSIZE](#), 75
[G_MAXUINT](#), 75
[G_MAXUINT16](#), 75
[G_MAXUINT32](#), 75
[G_MAXUINT64](#), 75
[G_MAXUINT8](#), 75
[G_MAXULONG](#), 75
[G_MAXUSHORT](#), 75
[g_message\(\)](#), 78
[g_message\(*formátumszöveg*,
 ...\)](#), 77
[G_MINDOUBLE](#), 75
[G_MINFLOAT](#), 75
[G_MININT](#), 75
[G_MININT16](#), 75
[G_MININT32](#), 75
[G_MININT64](#), 75
[G_MININT8](#), 75
[G_MINLONG](#), 75
[G_MINSHORT](#), 75
[g_new\(\)](#), 80
[g_new\(*típus*, *darabszám*\)](#), 79
[g_new0\(*típus*, *darabszám*\)](#), 80
[G_NORMALIZE_ALL](#), 95
[G_NORMALIZE_ALL_COMPOSE](#), 95
[G_NORMALIZE_DEFAULT](#), 94
[G_NORMALIZE_DEFAULT_COMPOSE](#),
 95
[G_OBJECT\(\)](#), 29
[g_object_get\(\)](#), 26, 28, 146,
 237, 271, 276, 305, 311
[g_object_get_data\(\)](#), 66–68
[g_object_new\(\)](#), 134, 146
[g_object_ref\(\)](#), 228
[g_object_set\(\)](#), 27, 28, 146,
 237, 271, 276, 305, 326
[g_object_set_data\(\)](#), 66–68,
 283
[g_object_set_data_full\(\)](#), 66
[g_object_set_full\(\)](#), 326
[g_object_set_property\(\)](#), 29
[g_object_unref\(\)](#), 140, 228
[G_OPTION_ARG_CALLBACK](#), 119
[G_OPTION_ARG_FILENAME](#), 119
[G_OPTION_ARG_FILENAME_ARRAY](#),
 119
[G_OPTION_ARG_INT](#), 119
[G_OPTION_ARG_NONE](#), 118, 119
[G_OPTION_ARG_STRING](#), 119
[G_OPTION_ARG_STRING_ARRAY](#),
 119
[g_option_context_add_main_entries\(\)](#),
 121
[g_option_context_free\(\)](#), 120
[g_option_context_new\(\)](#), 120
[g_option_context_parse\(\)](#), 120
[g_option_context_set_ignore_unknown_options\(\)](#),
 121
[G_OPTION_FLAG_HIDDEN](#), 118
[G_OPTION_FLAG_IN_MAIN](#), 118
[G_OPTION_FLAG_REVERSE](#), 118
[G_OS_BEOS](#), 76
[G_OS_UNIX](#), 76
[G_OS_WIN32](#), 75
[g_path_get_basename\(\)](#), 115,
 116
[g_path_get_dirname\(\)](#), 116

<code>g_path_is_absolute()</code> , 116	<code>g_string_truncate()</code> , 101
<code>g_printf()</code> , 83	<code>g_strjoin()</code> , 86
<code>g_realloc()</code> , 79, 80	<code>g_strndup()</code> , 82
<code>g_return_if_fail(<i>kifejezés</i>)</code> , 77	<code>g_strnfill()</code> , 82
<code>g_return_if_reached()</code> , 77	<code>g_strreverse()</code> , 83
<code>g_return_val_if_fail(<i>kifejezés</i>, <i>érték</i>)</code> , 77	<code>g_strrstr()</code> , 82
<code>g_return_val_if_reached(<i>érték</i>)</code> , 77	<code>g_strlen()</code> , 82
<code>g_snprintf()</code> , 83	<code>g_strsplit()</code> , 84, 85
<code>g_sprintf()</code> , 83	<code>g_strsplit_set()</code> , 85
<code>g_stpcpy()</code> , 82	<code>g_strstr_len()</code> , 82
<code>g_str_has_prefix()</code> , 82	<code>g_strstrip(<i>szöveg</i>)</code> , 84
<code>g_str_has_suffix()</code> , 82	<code>g_tree_destroy()</code> , 114
<code>g_strcanon()</code> , 84	<code>g_tree_foreach()</code> , 113, 114
<code>g_strchomp()</code> , 83, 84	<code>g_tree_height()</code> , 113
<code>g_strchug()</code> , 83, 84	<code>g_tree_insert()</code> , 112, 113
<code>g_strcompress()</code> , 84	<code>g_tree_lookup()</code> , 113
<code>g_strconcat()</code> , 85, 86	<code>g_tree_new()</code> , 110, 111
<code>g_strdup()</code> , 81, 166	<code>g_tree_new_full()</code> , 111, 112
<code>g_strdup_printf()</code> , 82	<code>g_tree_new_with_data()</code> , 111
<code>g_strescape()</code> , 84	<code>g_tree_nnodes()</code> , 113
<code>g_strfreev()</code> , 85	<code>g_tree_remove()</code> , 114
<code>g_string_append()</code> , 99	<code>g_tree_replace()</code> , 112
<code>g_string_append_c()</code> , 99, 100	<code>g_try_malloc()</code> , 79
<code>g_string_append_printf()</code> , 99	<code>g_try_realloc()</code> , 79
<code>g_string_append_unichar()</code> , 99, 100	<code>G_TYPE_BOOLEAN</code> , 263
<code>g_string_assign()</code> , 98	<code>G_TYPE_CHECK_INSTANCE_CAST()</code> , 132
<code>g_string_erase()</code> , 101	<code>G_TYPE_CHECK_INSTANCE_TYPE()</code> , 132
<code>g_string_free()</code> , 101	<code>g_type_class_add_private()</code> , 147
<code>g_string_insert()</code> , 100	<code>G_TYPE_DOUBLE</code> , 263
<code>g_string_insert_c()</code> , 100	<code>G_TYPE_INSTANCE_GET_PRIVATE()</code> , 147
<code>g_string_insert_unichar()</code> , 100	<code>G_TYPE_INT</code> , 29
<code>g_string_new()</code> , 98	<code>G_TYPE_INT</code> , 263
<code>g_string_new_len()</code> , 98	<code>G_TYPE_POINTER</code> , 263
<code>g_string_prepend()</code> , 99	<code>g_type_register_static()</code> , 138, 139
<code>g_string_prepend_c()</code> , 100	<code>G_TYPE_STRING</code> , 263
<code>g_string_prepend_unichar()</code> , 100	<code>G_UNICODE_BREAK_AFTER</code> , 89
<code>g_string_printf()</code> , 98, 99	<code>G_UNICODE_BREAK_ALPHABETIC</code> , 89
<code>g_string_sized_new()</code> , 98	<code>G_UNICODE_BREAK_AMBIGUOUS</code> , 89

356

G_UNICODE_BREAK_BEFORE, 89	G_UNICODE_BREAK_QUOTATION, 89
G_UNICODE_BREAK_BEFORE_AND_AFTER, 89	G_UNICODE_BREAK_SPACE, 89
G_UNICODE_BREAK_CARRIAGE_RETURN, 89	G_UNICODE_BREAK_SURROGATE, 89
G_UNICODE_BREAK_CLOSE_PUNCTUATION, 89	G_UNICODE_BREAK_SYMBOL, 89
G_UNICODE_BREAK_COMBINING_MARK, 89	G_UNICODE_BREAK_UNKNOWN, 89
G_UNICODE_BREAK_COMPLEX_CONTEXT, 89	G_UNICODE_BREAK_WORD_JOINER, 89
G_UNICODE_BREAK_CONTINGENT, 89	G_UNICODE_BREAK_ZERO_WIDTH_SPACE, 89
G_UNICODE_BREAK_EXCLAMATION, 89	G_UNICODE_CLOSE_PUNCTUATION, 89
G_UNICODE_BREAK_HANGUL_L_JAMO, 89	G_UNICODE_COMBINING_MARK, 89
G_UNICODE_BREAK_HANGUL_LV_SYLLABLE, 89	G_UNICODE_CONNECT_PUNCTUATION, 89
G_UNICODE_BREAK_HANGUL_LVT_SYLLABLE, 89	G_UNICODE_CURRENCY_SYMBOL, 89
G_UNICODE_BREAK_HANGUL_T_JAMO, 89	G_UNICODE_DASH_PUNCTUATION, 89
G_UNICODE_BREAK_HANGUL_V_JAMO, 89	G_UNICODE_DECIMAL_NUMBER, 89
G_UNICODE_BREAK_HYPHEN, 89	G_UNICODE_ENCLOSING_MARK, 89
G_UNICODE_BREAK_IDEOGRAPHIC, 89	G_UNICODE_FINAL_PUNCTUATION, 89
G_UNICODE_BREAK_INSEPARABLE, 89	G_UNICODE_FORMAT, 88
G_UNICODE_BREAK_INFIX_SEPARATOR, 89	G_UNICODE_INITIAL_PUNCTUATION, 89
G_UNICODE_BREAK_LINE_FEED, 89	G_UNICODE_LETTER_NUMBER, 89
G_UNICODE_BREAK_MANDATORY, 89	G_UNICODE_LINE_SEPARATOR, 89
G_UNICODE_BREAK_NEXT_LINE, 89	G_UNICODE_LOWERCASE_LETTER, 88
G_UNICODE_BREAK_NON_BREAKING_GLUE, 89	G_UNICODE_MATH_SYMBOL, 89
G_UNICODE_BREAK_NON_STARTER, 89	G_UNICODE_MODIFIER_LETTER, 88
G_UNICODE_BREAK_NUMERIC, 89	G_UNICODE_MODIFIER_SYMBOL, 89
G_UNICODE_BREAK_OPEN_PUNCTUATION, 89	G_UNICODE_NON_SPACING_MARK, 89
G_UNICODE_BREAK_POSTFIX, 89	G_UNICODE_OPEN_PUNCTUATION, 89
G_UNICODE_BREAK_PREFIX, 89	G_UNICODE_OTHER_LETTER, 88
	G_UNICODE_OTHER_NUMBER, 89
	G_UNICODE_OTHER_PUNCTUATION, 89
	G_UNICODE_OTHER_SYMBOL, 89
	G_UNICODE_PARAGRAPH_SEPARATOR, 89
	G_UNICODE_PRIVATE_USE, 88
	G_UNICODE_SPACE_SEPARATOR, 89

G_UNICODE_SURROGATE, 88	g_utf8_pointer_to_offset(), 91
G_UNICODE_TITLECASE_LETTER, 89	g_utf8_prev_char(), 91
G_UNICODE_UNASSIGNED, 88	g_utf8_strchr(), 92
G_UNICODE_UPPERCASE_LETTER, 89	g_utf8_strdown(), 94
g_unichar_break_type(), 89	g_utf8_strlen(), 92
g_unichar_digit_value(), 88	g_utf8_strncpy(), 92
g_unichar_isalnum(), 87	g_utf8_strrchr(), 92
g_unichar_isalpha(), 87	g_utf8_strreverse(), 93
g_unichar_iscntrl(), 87	g_utf8_strup(), 93
g_unichar_isdefined(), 88	g_utf8_validate(), 93
g_unichar_isdigit(), 87	g_value_set_int(), 29
g_unichar_isgraph(), 87	g_warning(), 78
g_unichar_islower(), 87	g_warning(<i>formátumszöveg</i> , ...), 78
g_unichar_isprint(), 87	gboolean, 58, 73, 119, 237–246, 271, 273–276, 280, 282, 283, 305, 306
g_unichar_ispunct(), 87	gchar, 74, 81
g_unichar_isspace(), 88	gchar *, 89, 119
g_unichar_istitle(), 88	gchar **, 119
g_unichar_isupper(), 88	gchararray, 237–241, 271, 273–275, 280, 287, 306
g_unichar_iswide(), 88	GCompareDataFunc(), 111
g_unichar_isxdigit(), 88	GCompareFunc(), 104, 111
g_unichar_tolower(), 88	gconstpointer, 74
g_unichar_totitle(), 88	GDK_BUTTON_PRESS, 59
g_unichar_toupper(), 88	gdk_color_parse(), 195
g_unichar_type(), 88	gdk_pixbuf_new_from_file(), 263
g_unichar_validate(), 87	GDK_SELECTION_CLIPBOARD, 250
g_unichar_xdigit_value(), 88	GDK_SELECTION_PRIMARY, 250
G_UNICODE_CONTROL, 88	GDK_TYPE_PIXBUF, 263
g_utf8_casefold(), 94, 95	GdkColor, 195, 237, 239, 241, 271, 273, 274
g_utf8_collate(), 94–96	GdkEventButton, 58
g_utf8_collate_key(), 96	GdkPixbuf, 280
g_utf8_collate_key_for_filename(), 96	GdkPixmap, 238, 239
g_utf8_find_next_char(), 92	gdoube, 75, 242, 243, 275
g_utf8_find_prev_char(), 91	GError, 120, 127
g_utf8_get_char(), 90	gettext(), 348
g_utf8_get_char_validated(), 90	gettext, 349
g_utf8_next_char(), 90, 97	gfloat, 75, 272, 305
g_utf8_normalize(), 94	
g_utf8_offset_to_pointer(), 90	

358

[gint](#), 74, 119, 239–243, 245, 271, 272, 275, 276, 287, 306
[gint16](#), 74
[gint32](#), 74
[gint64](#), 75
[gint8](#), 74
[glade](#), 11
[glade-2](#), 11
[GList](#), 102
[glong](#), 74
[GNode](#), 109, 110
[gnome-theme-manager](#), 33
[gnome-ui-properties](#), 53
[GnomeAppBar](#), 208
[GObject](#), 65, 66, 68, 132–134, 140, 270, 276
[gobject_class](#), 139
[GObjectClass](#), 140
[gombok](#), 184
[GOptionEntry](#), 118
[gpointer](#), 74
[green](#), 195
[gshort](#), 74
[gsize](#), 75
[gssize](#), 75
[GString](#), 97, 98, 101
[gtk-demo](#), 9
[gtk_button_set_label\(\)](#), 40
[GTK_BUTTONS_CANCEL](#), 184
[GTK_BUTTONS_CLOSE](#), 184
[GTK_BUTTONS_OK](#), 184
[GTK_BUTTONS_OK_CANCEL](#), 184
[GTK_BUTTONS_YES_NO](#), 184
[GTK_CELL_LAYOUT\(\)](#), 297
[gtk_cell_layout_add_attribute\(\)](#), 289, 297
[gtk_cell_layout_pack_end\(\)](#), 288
[gtk_cell_layout_pack_start\(\)](#), 288, 297
[gtk_cell_layout_set_attributes\(\)](#), 288, 289
[gtk_cell_layout_set_cell_data_func\(\)](#), 290
[gtk_cell_renderer_pixbuf_new\(\)](#), 280
[gtk_cell_renderer_progress_new\(\)](#), 287
[gtk_cell_renderer_text_new\(\)](#), 272
[gtk_cell_renderer_toggle_get_radio\(\)](#), 282
[gtk_cell_renderer_toggle_new\(\)](#), 282
[gtk_cell_renderer_toggle_set_radio\(\)](#), 282
[gtk_clipboard_get\(\)](#), 250
[gtk_color_selection_dialog_new\(\)](#), 195
[gtk_color_selection_get_current_alpha\(\)](#), 196
[gtk_color_selection_get_current_color\(\)](#), 196
[gtk_color_selection_set_current_alpha\(\)](#), 196
[gtk_color_selection_set_current_color\(\)](#), 196
[gtk_color_selection_set_has_opacity_control\(\)](#), 195
[gtk_combo_box_append_text\(\)](#), 47
[gtk_combo_box_get_active\(\)](#), 49
[gtk_combo_box_get_active_iter\(\)](#), 294
[gtk_combo_box_get_active_text\(\)](#), 48
[gtk_combo_box_insert_text\(\)](#), 47
[gtk_combo_box_new\(\)](#), 294
[gtk_combo_box_new_text\(\)](#), 293, 294
[gtk_combo_box_prepend_text\(\)](#), 48
[gtk_combo_box_remove_text\(\)](#), 48
[gtk_combo_box_set_active\(\)](#), 48

<code>gtk_combo_box_set_model()</code> ,	58
294	<code>GTK_ICON_SIZE_BUTTON</code> , 281
<code>gtk_dialog_add_buttons()</code> , 185	<code>GTK_ICON_SIZE_DIALOG</code> , 281
<code>GTK_DIALOG_DESTROY_WITH_PARENT</code> ,	<code>GTK_ICON_SIZE_DND</code> , 281
184	<code>GTK_ICON_SIZE_INVALID</code> , 281
<code>gtk_dialog_run()</code> , 185, 186,	<code>GTK_ICON_SIZE_LARGE_TOOLBAR</code> ,
188–190	281
<code>gtk_editable_select_region()</code> ,	<code>GTK_ICON_SIZE_MENU</code> , 281
45	<code>GTK_ICON_SIZE_SMALL_TOOLBAR</code> ,
<code>gtk_entry_comple-</code>	281
<code>tion_set_text_co-</code>	<code>gtk_icon_view_get_cursor()</code> ,
<code>lumn()</code> , 268	337
<code>GTK_ENTRY()</code> , 45	<code>gtk_icon_view_get_item_at_pos()</code> ,
<code>gtk_entry_completion_new()</code> ,	339
268	<code>gtk_icon_view_get_markup_column()</code> ,
<code>gtk_entry_completion_set_model()</code> ,	332
268	<code>gtk_icon_view_get_model()</code> ,
<code>gtk_entry_get_text()</code> , 44	332
<code>gtk_entry_set_completion()</code> ,	<code>gtk_icon_view_get_orientation()</code> ,
268	333
<code>gtk_entry_set_text()</code> , 44	<code>gtk_icon_view_get_path_at_pos()</code> ,
<code>gtk_exit()</code> , 20	338
<code>GTK_FILE_CHOOSER_ACTION_CREATE_FOLDER</code> ,	<code>gtk_icon_view_get_pixbuf_column()</code> ,
188	333
<code>GTK_FILE_CHOOSER_ACTION_OPEN</code> ,	<code>gtk_icon_view_get_selected_items()</code> ,
188	336
<code>GTK_FILE_CHOOSER_ACTION_SAVE</code> ,	<code>gtk_icon_view_get_selection_mode()</code> ,
188	335
<code>GTK_FILE_CHOOSER_ACTION_SELECT_FOLDER</code> ,	<code>gtk_icon_view_get_text_column()</code> ,
188	332
<code>gtk_file_chooser_dialog_new()</code> ,	<code>gtk_icon_view_item_activated()</code> ,
188	339
<code>gtk_file_chooser_get_filename()</code> ,	<code>gtk_icon_view_path_is_selected()</code> ,
190	336
<code>gtk_file_chooser_set_current_folder()</code> ,	<code>gtk_icon_view_select_all()</code> ,
189	336
<code>gtk_font_selection_dialog_get_font_name()</code> ,	<code>gtk_icon_view_select_path()</code> ,
192	336
<code>gtk_font_selection_dialog_new()</code> ,	<code>gtk_icon_view_selected_foreach()</code> ,
191	335
<code>gtk_font_selection_dialog_set_font_name()</code> ,	<code>gtk_icon_view_set_column_spacing()</code> ,
192	334
<code>gtk_frame_set_label()</code> , 64	<code>gtk_icon_view_set_cursor()</code> ,
<code>gtk_get_current_event_time()</code> ,	337

360

gtk_icon_view_set_item_width(), 124
 328
 gtk_icon_view_set_margin(), 326
 334
 gtk_icon_view_set_markup_column(), 205
 332
 gtk_icon_view_set_model(), 184
 327
 gtk_icon_view_set_orientation(), 184
 333
 gtk_icon_view_set_pixbuf_column(), 333
 328, 333
 gtk_icon_view_set_reorderable(), 333
 334
 gtk_icon_view_set_row_spacing(), 189
 333
 gtk_icon_view_set_selection_mode(), 185
 334
 gtk_icon_view_set_spacing(), 189
 333
 gtk_icon_view_set_text_column(), 189
 328, 332
 gtk_icon_view_unselect_all(), 312, 335
 337
 gtk_icon_view_unselect_path(), 312, 334
 336
 gtk_init(), 124
 gtk_init_check(), 124
 gtk_init_with_args(), 124, 125, 127, 128
 GTK_IS_LIST_STORE(), 311
 GTK_IS_TREE_STORE(), 311
 GTK_JUSTIFY_CENTER, 240
 GTK_JUSTIFY_FILL, 240
 GTK_JUSTIFY_LEFT, 240
 GTK_JUSTIFY_RIGHT, 240
 gtk_label_set_text(), 36
 gtk_list_store_append(), 264, 266, 269
 gtk_list_store_clear(), 263
 gtk_list_store_new(), 263, 269
 gtk_list_store_prepend(), 264
 gtk_list_store_set(), 264, 266, 269
 gtk_main(), 124
 gtk_main_iteration(), 208
 gtk_menu_popup(), 57, 60, 326
 gtk_menu_tool_button_set_menu(), 205
 gtk_message_dialog_new(), 184
 GTK_MESSAGE_ERROR, 184
 GTK_MESSAGE_INFO, 184
 GTK_MESSAGE_QUESTION, 184
 GTK_MESSAGE_WARNING, 184
 GTK_ORIENTATION_HORIZONTAL, 333
 GTK_ORIENTATION_VERTICAL, 333
 GTK_RESPONSE_APPLY, 189
 GTK_RESPONSE_CANCEL, 185, 189
 GTK_RESPONSE_CLOSE, 185, 189
 GTK_RESPONSE_DELETE_EVENT, 185
 GTK_RESPONSE_HELP, 189
 GTK_RESPONSE_NO, 185, 189
 GTK_RESPONSE_OK, 185, 189
 GTK_RESPONSE_YES, 185, 189
 GTK_SELECTION_BROWSE, 312, 335
 GTK_SELECTION_MULTIPLE, 312, 335
 GTK_SELECTION_NONE, 312, 334
 GTK_SELECTION_SINGLE, 312, 335
 GTK_SORT_ASCENDING, 305, 306
 GTK_SORT_DESCENDING, 305, 306
 gtk_spin_button_get_value(), 46
 gtk_spin_button_get_value_as_int(), 46
 gtk_spin_button_set_value(), 45
 GTK_STOCK_APPLY, 188
 GTK_STOCK_CANCEL, 189
 GTK_STOCK_HELP, 189
 GTK_STOCK_NEW, 189
 GTK_STOCK_NO, 189
 GTK_STOCK_OK, 189
 GTK_STOCK_OPEN, 189
 GTK_STOCK_PRINT, 189
 GTK_STOCK_REMOVE, 189

GTK_STOCK_SAVE, 189	gtk_text_buffer_insert_with_tags_by_name(),
GTK_STOCK_YES, 189	232
gtk_text_buffer_apply_tag(),	gtk_text_buffer_move_mark(),
233	229
gtk_text_buffer_apply_tag_by_name(),	gtk_text_buffer_move_mark_by_name(),
233, 234	229
gtk_text_buffer_copy_clipboard(),	gtk_text_buffer_paste_clipboard(),
251	251
gtk_text_buffer_create_mark(),	gtk_text_buffer_remove_all_tags(),
227	234
gtk_text_buffer_create_tag(),	gtk_text_buffer_remove_tag(),
231, 237	233
gtk_text_buffer_cut_clipboard(),	gtk_text_buffer_remove_tag_by_name(),
251	233
gtk_text_buffer_delete_mark(),	gtk_text_buffer_select_range(),
228, 230	247
gtk_text_buffer_delete_mark_by_name(),	gtk_text_buffer_set_modified(),
228, 230	254
gtk_text_buffer_delete_selection(),	gtk_text_buffer_set_text(),
250	211
gtk_text_buffer_get_end_iter(),	GTK_TEXT_DIR_LTR, 238
211	GTK_TEXT_DIR_NONE, 238
gtk_text_buffer_get_has_selection(),	GTK_TEXT_DIR_RTL, 238
247	gtk_text_iter_backward_char(),
gtk_text_buffer_get_insert(),	220
246	gtk_text_iter_backward_chars(),
gtk_text_buffer_get_iter_at_mark(),	220
228	gtk_text_iter_backward_cursor_position(),
gtk_text_buffer_get_mark(),	222
229, 246	gtk_text_iter_backward_cursor_positions(),
gtk_text_buffer_get_modified(),	222
254	gtk_text_iter_backward_find_char(),
gtk_text_buffer_get_selection_bound(),	225
247	gtk_text_iter_backward_line(),
gtk_text_buffer_get_selection_bounds(),	221
247	gtk_text_iter_backward_lines(),
gtk_text_buffer_get_start_iter(),	221
211	gtk_text_iter_backward_search(),
gtk_text_buffer_get_text(),	226
212	gtk_text_iter_backward_sentence_start(),
gtk_text_buffer_insert(),	223
212, 232, 233	gtk_text_iter_backward_sentence_starts(),
gtk_text_buffer_insert_with_tags(),	223
232	gtk_text_iter_backward_to_tag_toggle(),

362

```

224          gtk_text_iter_forward_word_end(),
gtk_text_iter_backward_word_start(), 221, 222
221          gtk_text_iter_forward_word_ends(),
gtk_text_iter_backward_word_starts(), 222
222          gtk_text_iter_forward_word_start(),
gtk_text_iter_begins_tag(), 222
218          gtk_text_iter_get_buffer(),
gtk_text_iter_compare(), 226 217
gtk_text_iter_ends_line(), gtk_text_iter_get_bytes_in_line(),
219 220
gtk_text_iter_ends_sentence(), gtk_text_iter_get_char(), 217
219          gtk_text_iter_get_chars_in_line(),
gtk_text_iter_ends_tag(), 218 220
gtk_text_iter_ends_word(), gtk_text_iter_get_line(), 217
219          gtk_text_iter_get_line_index(),
gtk_text_iter_equal(), 226 217
gtk_text_iter_forward_char(), gtk_text_iter_get_line_offset(),
220, 222 217
gtk_text_iter_forward_chars(), gtk_text_iter_get_marks(),
220 217
gtk_text_iter_forward_cursor_position(), gtk_text_iter_get_offset(),
222 217
gtk_text_iter_forward_cursor_position_to_tag(), 219
222          gtk_text_iter_get_text(), 217
gtk_text_iter_forward_find_char(), gtk_text_iter_get_toggled_tags(),
224, 225 218
gtk_text_iter_forward_line(), gtk_text_iter_has_tag(), 218
220, 221          gtk_text_iter_in_range(), 226
gtk_text_iter_forward_lines(), gtk_text_iter_inside_sentence(),
221 219
gtk_text_iter_forward_search(), gtk_text_iter_inside_word(),
225, 226 219
gtk_text_iter_forward_sentence_end(), gtk_text_iter_is_cursor_position(),
223 219
gtk_text_iter_forward_sentence_end_to_tag(), 220
223          gtk_text_iter_is_start(), 220
gtk_text_iter_forward_sentence_start(), 226
223          gtk_text_iter_order(), 226
gtk_text_iter_forward_to_end(), gtk_text_iter_set_line(), 223
224          gtk_text_iter_set_line_index(),
223
gtk_text_iter_forward_to_line_end(), gtk_text_iter_set_line_offset(),
224 223
gtk_text_iter_forward_to_tag_toggle(), 223
224          gtk_text_iter_set_offset(),
223

```

gtk_text_iter_starts_line(),	gtk_tree_model_get_iter(),
219	258
gtk_text_iter_starts_sentence(),	gtk_tree_model_get_iter_first(),
219	255
gtk_text_iter_starts_word(),	gtk_tree_model_get_iter_from_string(),
219	255
gtk_text_iter_toggles_tag(),	gtk_tree_model_get_n_columns(),
218	261
gtk_text_mark_get_buffer(),	gtk_tree_model_get_path(),
229	258
gtk_text_mark_get_deleted(),	gtk_tree_model_get_string_from_iter(),
229	257
gtk_text_mark_get_name(),	gtk_tree_model_iter_children(),
230	256
gtk_text_mark_set_visible(),	gtk_tree_model_iter_has_child(),
230	
gtk_text_view_get_buffer(),	gtk_tree_model_iter_n_children(),
211	
gtk_text_view_move_mark_onscreen(),	256
230, 231	gtk_tree_model_iter_next(),
gtk_text_view_scroll_mark_onscreen(),	256
231	gtk_tree_model_iter_nth_child(),
GTK_TOGGLE_BUTTON(),	51
257	
gtk_toggle_button_get_active(),	gtk_tree_model_iter_parent(),
42, 51	257
gtk_toggle_button_get_inconsistent(),	gtk_tree_path_append_index(),
42	259
gtk_toggle_button_set_active(),	gtk_tree_path_compare(),
41, 51	260
gtk_toggle_button_set_inconsistent(),	gtk_tree_path_copy(),
42	259
	gtk_tree_path_down(),
	260
gtk_toggle_tool_button_get_active(),	gtk_tree_path_free(),
313, 318, 323, 336	258,
205	gtk_tree_path_get_depth(),
	259
gtk_toggle_tool_button_set_active(),	259
205	gtk_tree_path_is_ancestor(),
	260
gtk_tool_button_set_icon_widget(),	205
205	gtk_tree_path_new(),
	258
gtk_tool_button_set_label(),	gtk_tree_path_new_first(),
205	259
GTK_TREE_MODEL(),	255
gtk_tree_model_foreach(),	gtk_tree_path_new_from_string(),
261, 262	259
gtk_tree_model_get(),	261
	gtk_tree_path_next(),
	260
gtk_tree_model_get_column_type(),	gtk_tree_path_prepend_index(),
261	259
	gtk_tree_path_prev(),
	260

364

gtk_tree_path_to_string(),	gtk_tree_view_column_add_attribute(),
259	289
gtk_tree_path_up(),	260
gtk_tree_selection_count_selected_rows(),	GTK_TREE_VIEW_COLUMN_AUTOSIZE,
302, 306	
313	GTK_TREE_VIEW_COLUMN_FIXED,
gtk_tree_selection_get_mode(),	302, 306
312	gtk_tree_view_column_get_width(),
gtk_tree_selection_get_selected(),	302
312	GTK_TREE_VIEW_COLUMN_GROW_ONLY,
gtk_tree_selection_get_selected_rows(),	302, 306
313, 317	gtk_tree_view_column_new(),
gtk_tree_selection_iter_is_selected(),	299, 300
314	gtk_tree_view_column_new_with_att-
gtk_tree_selection_path_is_selected(),	tributes(), 299
314	gtk_tree_view_column_new_with_attributes(),
gtk_tree_selection_select_all(),	301
315	gtk_tree_view_column_set_alignment(),
gtk_tree_selection_select_iter(),	304
314	gtk_tree_view_column_set_clickable(),
gtk_tree_selection_select_path(),	303
313	gtk_tree_view_column_set_expand(),
gtk_tree_selection_select_range(),	303
315	gtk_tree_view_column_set_fixed_width(),
gtk_tree_selection_set_mode(),	302
312	gtk_tree_view_column_set_max_width(),
gtk_tree_selection_unselect_all(),	303
315	gtk_tree_view_column_set_min_width(),
gtk_tree_selection_unselect_iter(),	303
314	gtk_tree_view_column_set_reorderable(),
gtk_tree_selection_unselect_path(),	304
314	gtk_tree_view_column_set_resizable(),
gtk_tree_selection_unselect_range(),	302
315	gtk_tree_view_column_set_sizing(),
gtk_tree_store_append(),	269
302	
gtk_tree_store_clear(),	269
gtk_tree_store_new(),	268
304, 305	
gtk_tree_store_prepend(),	269
gtk_tree_store_set(),	269
305	
gtk_tree_view_append_column(),	gtk_tree_view_column_set_sort_order(),
299	305
gtk_tree_view_collapse_all(),	gtk_tree_view_column_set_spacing(),
318	301
gtk_tree_view_collapse_row(),	gtk_tree_view_column_set_title(),
319	303

gtk_tree_view_column_set_visible()	302	gtk_tree_view_set_hover_expand(),	309
gtk_tree_view_column_set_widget()	304	gtk_tree_view_set_hover_selection(),	309
gtk_tree_view_expand_all(),	318	gtk_tree_view_set_model(),	300, 311
gtk_tree_view_expand_row(),	318	gtk_tree_view_set_reorderable(),	310
gtk_tree_view_expand_to_path(),	318	gtk_tree_view_set_rules_hint(),	310
gtk_tree_view_get_column(),	319	gtk_tree_view_set_search_column(),	310
gtk_tree_view_get_columns(),	319	gtk_tree_view_set_search_entry(),	310
gtk_tree_view_get_cursor(),	317	gtk_widget_activate(),	71
gtk_tree_view_get_model(),	311	gtk_widget_destroy(),	68, 181,
gtk_tree_view_get_path_at_pos(),	322, 326	gtk_widget_get_name(),	72
gtk_tree_view_get_selection(),	311	gtk_widget_get_size_request(),	71
GTK_TREE_VIEW_GRID_LINES_BOTH,	309	gtk_widget_grab_focus(),	72
GTK_TREE_VIEW_GRID_LINES_HORIZONTAL,	308	gtk_widget_hide(),	69, 181
GTK_TREE_VIEW_GRID_LINES_NONE,	308	gtk_widget_hide_all(),	69
GTK_TREE_VIEW_GRID_LINES_VERTICAL,	309	gtk_widget_is_focus(),	71
gtk_tree_view_set_cursor(),	318	gtk_widget_set_name(),	72
gtk_tree_view_set_enable_search(),	308	gtk_widget_set_sensitive(),	70
gtk_tree_view_set_enable_tree_view(),	308	gtk_widget_set_size_request(),	71
gtk_tree_view_set_fixed_height(),	308	gtk_widget_show(),	69, 70, 180,
gtk_tree_view_set_grid_lines(),	308	gtk_widget_show_all(),	69, 181
gtk_tree_view_set_headers_clickable(),	309	gtk_window_set_title(),	61
gtk_tree_view_set_headers_visible(),	309	GTK_WRAP_CHAR,	245
		GTK_WRAP_NONE,	245, 246
		GTK_WRAP_WORD,	246
		GTK_WRAP_WORD_CHAR,	246
		GtkButton,	38, 41, 203, 205
		GtkCellLayout,	287, 294, 297,
			298, 338
		GtkCellRenderer,	270, 271
		GtkCellRendererPixbuf,	279
		GtkCellRendererText,	272, 276
		GtkCheckButton,	50

366

[GtkCheckMenuItem](#), 56
[GtkClipboard](#), 250
[GtkColorSelection](#), 195
[GtkColorSelectionDialog](#), 195
[GtkCombo](#), 46
[GtkComboBox](#), 47–49, 293, 294
[GtkComboBoxEntry](#), 47–49
[GtkContainer](#), 38
[GtkEditable](#), 45
[GtkEntry](#), 43, 45
[GtkEventButton](#), 60
[GtkFrame](#), 64
[GtkHBox](#), 64
[GtkIconView](#), 327, 338
[GtkImage](#), 37
[GtkJustification](#), 240
[GtkLabel](#), 35
[GtkListStore](#), 254, 255, 262, 327, 328
[GtkMenuBar](#), 206
[GtkMenuToolButton](#), 203
[GtkObject](#), 270
[GtkOptionMenu](#), 46
[GtkRadioButton](#), 52
[GtkRadioToolButton](#), 203
[GtkSeparatorToolItem](#), 203
[GtkSortType](#), 306
[GtkSpinButton](#), 45
[GtkTable](#), 64
[GtkTextBuffer](#), 210, 245
[GtkTextCharPredicate\(\)](#), 225
[GtkTextDirection](#), 238
[GtkTextIter](#), 210, 216, 226, 227, 255
[GtkTextMark](#), 210, 226, 227, 246, 255
[GtkTextTag](#), 210
[GtkTextView](#), 210, 245, 246
[GtkToggleButton](#), 41, 50
[GtkToggleToolButton](#), 203
[GtkToolBar](#), 202
[GtkToolButton](#), 203
[GtkTreeIter](#), 255, 258, 264
[GtkTreeModel](#), 255, 261, 327
[GtkTreePath](#), 255, 258, 313
[GtkTreeSelection](#), 311
[GtkTreeStore](#), 254, 255, 268
[GtkTreeViewColumn](#), 298, 319
[GtkTreeViewColumnSizing](#), 306
[GtkVBox](#), 63
[GtkWidget](#), 37, 47, 65, 68, 195, 207, 270, 294
[GtkWidget *](#), 180, 306
[GtkWindow](#), 61
[GtkWrapMode](#), 245
[GTraverseFunc\(\)](#), 114
[GTree](#), 110
[GType](#), 132, 261, 263
[GTypeInfo](#), 138
[guchar](#), 74
[guint](#), 74, 272, 281
[guint16](#), 74
[guint32](#), 75
[guint64](#), 75
[guint8](#), 74
[gulong](#), 74
[gunichar](#), 87
[gushort](#), 74
[GValue](#), 28, 29, 144
[gvim](#), 10, 23
[gzip](#), 21

[határ](#), 225
[HAVE_CONFIG_H](#), 347
[HOME](#), 116
[hossz](#), 212
[hu](#), 349

[ind](#), 30
[insert](#), 246, 249
[int](#), 74

[jelölőnégyzet](#), 51
[jelleg](#), 188

[könyvtárnév](#), 190
[kapcsolók](#), 184
[kapcsolók](#), 225
[kezdet](#), 45

kezdete, 225
 klass, 139
 kulcsfelszabadító, 112

 LANG, 348, 349
 last, 162
 LC_ALL, 348
 leave, 40
 len, 97
 libxml, 149
 libxml2, 149
 line, 162
 long, 74
 long long int, 75
 long_name, 118
 lookup_widget(), 30, 31, 35, 185

 main(), 85, 120, 124, 125, 127, 128
 make, 14, 16
 make dist, 341
 make distcheck, 341
 make install, 14
 Makefile, 153
 malloc(), 79, 80
 mark-set, 248, 249
 MAX(a, b), 76
 message, 127
 mező, 44, 45
 mező, 44
 MIN(a, b), 76
 move-cursor, 44

 név(), 248, 276, 283
 név, 30
 név, 227
 név1, 232
 name, 161
 new, 80
 next, 110, 162
 nodesetval, 176
 ns, 162
 NULL, 27, 30, 48, 66, 71, 79–82, 85, 86, 91, 92, 101, 102, 106, 107, 112, 113, 115, 121, 123, 125, 154–158, 162–165, 168, 169, 173, 174, 189, 190, 205, 218, 224, 225, 227, 229, 230, 232, 233, 237–241, 249, 251, 256, 257, 269, 288, 290, 291, 299, 310, 311, 313, 317–319, 323, 332, 337–339

 oszlop, 323

 PACKAGE, 347
 PACKAGE_DATA_DIR, 347
 PANGO_SCALE_LARGE, 243
 PANGO_SCALE_MEDIUM, 243
 PANGO_SCALE_SMALL, 243
 PANGO_SCALE_X_LARGE, 243
 PANGO_SCALE_X_SMALL, 243
 PANGO_SCALE_XX_LARGE, 243
 PANGO_SCALE_XX_SMALL, 243
 PANGO_STRETCH_CONDENSED, 243
 PANGO_STRETCH_EXPANDED, 243
 PANGO_STRETCH_EXTRA_CONDENSED, 243
 PANGO_STRETCH_EXTRA_EXPANDED, 243
 PANGO_STRETCH_NORMAL, 243
 PANGO_STRETCH_SEMI_CONDENSED, 243
 PANGO_STRETCH_SEMI_EXPANDED, 243
 PANGO_STRETCH_ULTRA_EXPANDED, 243
 PANGO_STRETCH_ULTRA_CONDENSED, 243
 PANGO_STYLE_ITALIC, 244
 PANGO_STYLE_NORMAL, 244
 PANGO_STYLE_OBLIQUE, 244
 PANGO_UNDERLINE_DOUBLE, 244
 PANGO_UNDERLINE_ERROR, 244
 PANGO_UNDERLINE_LOW, 244
 PANGO_UNDERLINE_NONE, 244

368

PANGO_UNDERLINE_SINGLE, 244
 PANGO_VARIANT_NORMAL, 245
 PANGO_VARIANT_SMALL_CAPS, 245
 PANGO_WEIGHT_BOLD, 245
 PANGO_WEIGHT_HEAVY, 245
 PANGO_WEIGHT_LIGHT, 245
 PANGO_WEIGHT_NORMAL, 245
 PANGO_WEIGHT_SEMIBOLD, 245
 PANGO_WEIGHT_ULTRABOLD, 245
 PANGO_WEIGHT_ULTRALIGHT, 245
 pango_attr_list_new(), 273
 PANGO_ELLIPSIZE_END, 274
 PANGO_ELLIPSIZE_MIDDLE, 274
 PANGO_ELLIPSIZE_NONE, 273
 PANGO_ELLIPSIZE_START, 274
 PANGO_SCALE, 242, 243
 pango_tab_array_new_with_positions(), 244
 PangoAttrList, 273
 PangoEllipsizeMode, 273
 PangoFontDescription, 239, 274
 PangoStretch, 243, 275
 PangoStyle, 244, 275
 PangoTabArray, 244
 PangoUnderline, 244, 275
 PangoVariant, 245, 276
 PangoWrapMode, 276
 parent, 207
 parent, 162
 parent_class, 141
 patch, 342, 344, 345
 PIP_IS_OBJECT(), 132, 133
 PIP_IS_OBJECT_CLASS(), 132
 PIP_OBJECT(), 132
 pip_object_class_init(), 139, 140
 PIP_OBJECT_GET_CLASS(), 133
 pip_object_get_type(), 132, 134, 138
 pip_object_new(), 134
 PIP_TYPE_OBJECT(), 138
 PIP_TYPE_OBJECT, 131, 132, 134
 PipObject, 129, 132–134, 139–141
 PipObjectClass, 140
 pkg-config, 152, 153
 PKG_CHECK_MODULES(), 152
 pressed, 40
 prev, 162
 printf(), 78, 82, 83, 99, 183–185
 priv, 148
 properties, 162, 163
 q_list_foreach(), 106
 q_list_insert_sorted(), 103
 q_list_position(), 107
 raktár, 262
 realize, 47–49, 248
 realloc(), 79
 red, 195
 released, 40
 return, 77
 rm, 117
 scp, 21
 selection_bound, 246
 short, 74
 short_name, 118
 show_menu, 205
 snprintf(), 83
 sprintf(), 83, 98
 static, 8
 str, 97
 strchr(), 92
 strcmp(), 96, 103, 260
 strdup(), 81
 stringval, 176
 strncpy(), 92
 struct _xmlAttr *next, 164
 struct _xmlAttr *prev, 164
 struct _xmlDoc *doc, 164
 struct _xmlNode *children, 164
 struct _xmlNode *last, 164
 struct _xmlNode *parent, 164
 szöveg, 44, 212, 225
 szülő, 184, 188

szmemória, 212, 227, 232
 típus, 184
 tar, 21, 341
 text(), 172
 text, 166, 168
 toggle-overwrite, 44
 toggled, 41, 204
 toggled, 51, 283, 286
 TreeView, 298
 TRUE, 42, 51, 58, 73, 76, 93, 101, 114, 121, 189, 205, 218, 220–227, 230, 231, 238, 247, 254–258, 262, 282, 288, 319, 322, 336
 type, 160–162, 164, 168, 169, 176
 uint16, 195
 unsigned char, 74
 unsigned int, 74
 unsigned long, 74
 unsigned short, 74
 vég, 45
 vége, 225
 VERSION, 347
 version, 150
 vim, 21
 visszatérési érték, 30, 44, 51, 125, 184, 185, 188, 189
 visszatérési érték, 190
 visszatérési_érték, 112, 225, 226
 void, 77
 void *, 74
 void *_private, 164
 window1, 180
 window2, 180
 x, 323
 xFree(), 174
 xml, 150, 151
 XML_ATTRIBUTE_NODE, 163, 164, 169
 XML_CDATA_SECTION_NODE, 161
 XML_COMMENT_NODE, 161
 XML_ELEMENT_NODE, 161, 168
 XML_TEXT_NODE, 161, 162, 164
 xmlAddChild(), 169
 xmlAddNextSibling(), 170
 xmlAddPrevSibling(), 170
 xmlAddSibling(), 170
 xmlAttr, 163, 164
 xmlAttrPtr, 163
 xmlCharStrdup(), 167
 xmlDoc, 160
 xmlDocPtr, 155, 160
 xmlElementType type, 164
 xmlEncodeEntitiesReentrant(), 157, 158
 xmlFree(), 157
 xmlFreeDoc(), 154
 xmlGetNodePath(), 173
 xmlKeepBlanksDefault(), 154
 xmlNewCDataBlock(), 169
 xmlNewChild(), 157, 158, 167
 xmlNewComment(), 168
 xmlNewDoc(), 154
 xmlNewDocNode(), 155, 167
 xmlNewNode(), 168
 xmlNewNodeEatName(), 168
 xmlNewText(), 168
 xmlNode, 160, 161, 163, 164
 xmlNodePtr, 160
 xmlNodeSet, 176
 xmlNodeSetPtr, 176
 xmlNs *ns, 164
 xmlParseFile(), 156
 xmlSaveFormatFileEnc(), 154
 xmlSetProp(), 158, 167
 xmlStrdup(), 167
 xmlUnsetProp(), 159
 xmlXPathCastToBoolean(), 174
 xmlXPathCastToNumber(), 174
 xmlXPathCastToString(), 174
 xmlXPathEvalExpression(), 173, 174
 xmlXPathFreeContext(), 173

370

`xmlXPathFreeObject()`, 173
`xmlXPathNewContext()`, 173
`xmlXPathObject`, 176
`xmlXPathObjectPtr`, 173–177
`XPATH_BOOLEAN`, 176
`XPATH_NODESET`, 176
`XPATH_NUMBER`, 176
`XPATH_STRING`, 176
`XPATH_UNDEFINED`, 176
`XPATH_XSLT_TREE`, 176

`y`, 323

Ajánlás

Kedves Olvasó!

Kérem, engedjen meg néhány szót e könyv létrejöttéről!

Cégünk a Blum Szoftver Mérnökség Kft. (www.blumsoft.com) 2005 óta foglalkozik nyílt forráskódú szoftverfejlesztéssel Maemo környezetben (www.maemo.org), melynek alapja a GTK+. A platform azóta szerkesztés része mindennapi fejlesztéseinknek és feltehetően szép jövő előtt áll többek között a beágyazott rendszerek (mobil eszközök, műszerek stb.) piacán. Sajnos magyar nyelven elég kevés irodalom található erről az izgalmas területről, így Ön egy hiánypótló művet „tart a kezében”. Pere László kollégánk egyéni munkájának támogatásával szeretnénk hazánkban is népszerűsíteni ezt a programozási környezetet. Ez úton szeretnék köszönetet mondani munkájáért és a lehetőségért, hogy szabadon letölthető formában rendelkezésre bocsátotta könyvét!

Üdvözlettel:

Blum László
Cégvezető

Veszprém, 2007 október 17