Egységes és objektumközpontú adatbázis-kezelés (2. rész)

A folytatásában a bemutatjuk, hogyan kezelhetünk Qt rendszer alatt SQL sormutatót, és készíthetünk grafikus felületet programoknak a Qt Designer segítségével.

Egy további adatkezelő osztály a `QSqlCursor`, amit közvetlen adatkezeléshez is használhatunk, de a fejlesztők fő célja az volt, hogy a grafikus elemek adatforrásának a szerepét is betölthesse. Ebben a pontban csak az adatkezelő vonatkozásokat tekintjük át, így példaprogramjaink továbbra is karakteres képernyőn fogják megjeleníteni a kimenetüket.
Ez az osztály lehetővé teszi egy táblára vagy nézetre (view) épülő SQL `select` parancs megalkotását és végrehajtását, amelynek az SQL sormutatónak nevezett eredménytábla lesz az eredménye. A sormutató sorain haladva adatmódosító utasításokat is kiadhatunk. A gyors megértés érdekében lássunk egy egyszerű példát! A feladat legyen az, hogy az `ujnevek` tábla azon sorait listázzuk ki a képernyőn, ahol a `honap=1` feltétel teljesül (lásd még az 55. CD Magazin/Qt/sql_3.cpp könyvtárában).

```
1. //
2. // sql_3.cpp
3. //
4. #include <iostream>
5. #include <qapplication.h>
6. #include <qsqldatabase.h>
7. #include <qsqlcursor.h>
8.
9. using std::cout;
10.
11.//--- A program indulási pontja ---
12.int main(int argc, char **argv)
13.{
14. QApplication app(argc, argv, false);
15. QSqlDatabase *pg = QSqlDatabase::addDatabase
    ➥( "QPSQL7", "PG_CS_ADATOK" );
16. if ( !pg ) { cout <<
    ➥"Hiba az illesztőprogram betöltésekor!";
    ➥return (1); }
17.
18. pg->setDatabaseName("cs_adatok");
19. pg->setUserName("postgres");
20. pg->setPassword("111111");
21. pg->setHostName("localhost");
22. pg->setPort( 5432 );
23.
24. if ( !pg->open() )
25. {
26. cout << "Hiba: az adatbázis nem nyitható
    ➥meg!";
27. return (1);
28. }
29.
30. QSqlCursor cur("ujnevek", true, pg);
31. cur.select("honap='1'");
32.
33. while ( cur.next() )
34. {
35. cout << "\n"
    ➥<< cur.value( "honap" ).toString() << "\t";
36. cout
    ➥<< cur.value( "nap" ).toString() << "\t";
37. cout << cur.value( "nevnap" ).toString();
38. }
39.
40. pg->close();
41. return 0;
42.}
```

Az 1–29. sor ismert fogalmakat tartalmaz. A 30. sorban egy `cur` nevű `QSqlCursor` típusú objektumot határoztunk meg. A létrehozó első értékében megmondtuk, hogy a végrehajtandó `select` az `ujnevek` táblára épüljön. A második, `true` érték azt kéri, hogy önműködően jöjjön létre a sorokat és a mezőket elérhetővé tevő `QSqlRecord` és `QSqlField` objektumegyüttes; a harmadik érték tisztázza azt, hogy melyik adatbázis-kapcsolaton szeretnénk dolgozni. A 31. sorban történik a `select` parancs véglegesítése és végrehajtása.
A `select()` tagfüggvénynek több alakja is van, így megadhatunk szűrőket (esetünkben `honap='1'`), sorrendeket, illetve mindkettőt egyszerre. A fentiek alapján létrehozott, ténylegesen végrehajtott SQL-parancs a következő lesz:

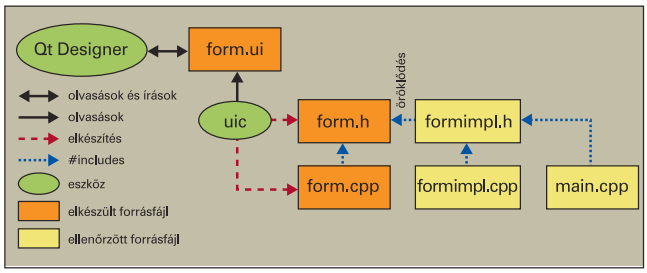
```
select * from ujnevek where honap='1';
```

A 33–38. sorban egy ciklusban kiírjuk a lekérdezett adatokat. A `cur.value` egy `QSqlRecord` objektumra mutat, amelyen keresztül az oszlopok nevét (használhatnánk indexet is) használva könnyen hozzáférhetünk adatainkhoz.
A fenti program által kiírt lista nem biztos, hogy a napokat nagyság szerinti sorrendben hozza le. Amennyiben ezt a feladatot is meg szeretnénk oldani, akkor a `cur.select()` hívást a következőképpen át kell fogalmazni:

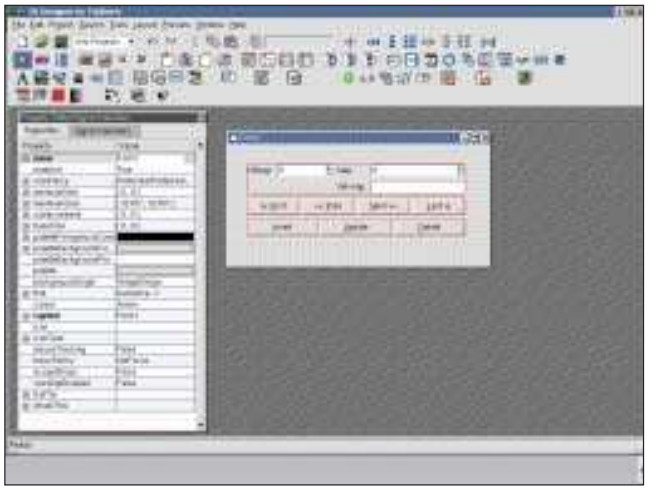
```
...
QSqlIndex idIndex;
QStringList ind = QStringList() << "nap";
idIndex = cur.index( ind );
cur.select("honap='1'", idIndex);
...
```

Látható, hogy a szűrőfeltételt a `select()` első értékében meghagytuk. A második értéket egy `QSqlIndex` objektum segítségével lehet megadni, ennek létrehozását a sormutatóobjektum az `index()` tagfüggvénnyel, a fenti módon támogatja. A létrehozott és végrehajtott SQL-parancs ez esetben a következő lesz:

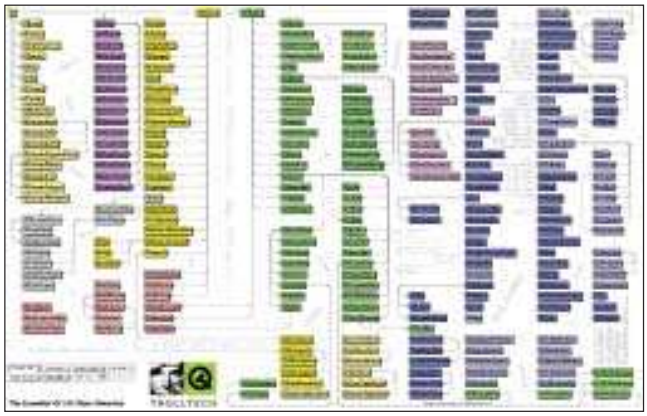
```
select * from ujnevek where honap='1' order
by nap;
```



A C++-forráskód létrehozása



A Qt Designer



A Qt osztálydiagram

Most már tudjuk, hogyan hozhatunk létre egy SQL sormutatót – a következőkben annak módját ismerjük meg, hogyan lehet a sormutató használatával adatmódosító műveleteket végrehajtani. Már az elején felhívjuk a figyelmet a `QSqlCursor::setPrimaryIndex()` tagfüggvényre, ami arról ad tájékoztatást, hogyan lehet egy táblában vagy nézetben egy sort egyértelműen azonosítani. Ezt a hívást az `update` és `delete` műveletek igénylik, az `insert` enélkül is hibátlanul működik. Amennyiben a táblának van elsődleges kulcsa és ez csak egyetlen mezőből áll, ez a hívás elhagyható. Emlékezzünk vissza, hogy egy sormutatónaknak mindig van egy pillanatnyi sora, ami egyébként `QSqlRecord` típusú. Ez a sor egy átmenetitár-területen van létrehozva, aminek a címét a `QSqlCursor` `primeInsert()`, `primeUpdate()` vagy `primeDelete()` tagfüggvényével le is kérhetjük, attól függően, hogy mit szeretnénk csinálni.
A `QSqlCursor` osztály kalkulált mezőket is tartalmazhat. Nézzük meg az alábbi programrészletet, ami az `update` műveletet mutatja be; a program az `i.Imrel.névnap` értéket `irImre(én)le` értékre cseréli ki.

```
1. ...
2. QSqlCursor cur("ujnevek", true, pg);
3. QSqlIndex idIndex;
4. QStringList ind = QStringList()
                ➥<< "honap" << "nap";
5. idIndex = cur.index( ind );
6. // A következő _ sor fontos a del()
   // és az update() működéséhez
7. cur.setPrimaryIndex( idIndex );
8. cur.select( "nevnap='Imre'" );
9. cur.next();
10.QSqlRecord *rec = cur.primeUpdate();
   // Csak lekérjük
11. rec->setValue("nevnap", "Imre (én)");
12. cout << "\nRekordszám: " << cur.update();
13....
```

A fenti kódrészlet a szokásos adatbázis-kapcsolódással és az alkalmazásobjektum létrehozásával indul, amit az 1. sorban `io...` io-tal jeleztünk. Az SQL `update` művelet szempontjából érdekes kódrészletet a 2–12. sorban emeltük ki. A 2. sorban létrehoztunk egy `cur` sormutatóobjektumot az `ujnevek` táblára. A 3–5. sor egy olyan `QSqlIndex` objektum (melynek neve `idIndex`) létrehozását mutatja, amiben az `ujnevek` tábla `honap` és `nap` oszlopai az indexelés szempontjai. A `cur` objektum `index()` tagfüggvénye szolgáltatja az indexelőobjektumunk létrehozását, amit ezután a 7. sorban a `setPrimaryIndex()` hívásban használni tudunk, így megteremtettük az `update` művelet használhatóságát. A 8. sor lefuttatja az SQL `select` lekérdezést a `nevnap='Imre'` szűrés mellett. A 10. sorban a `rec` rekordmutatót az `update` átmeneti tár memóriacímre állítjuk, így bármelyik is legyen a sormutató pillanatnyi sora, a `rec` is eléri azt mivel `QSqlRecord` osztálybeli objektum. A `rec` mutatón keresztül könnyen módosítható a sormutató pillanatnyi sora. A `setValue()` tagfüggvény első értéke az oszlopnév, míg a második tulajdonság (argumentum) a beállítandó új érték. A 12. sor lényegi része a `cur.update()` hívás, ami egy SQL `update` parancs kiadását jelenti, illetve visszaadja a sikeresen módosított sorok számát.
Az SQL sormutató használatával végrehajtható törlés művelet is egyszerű. Ebben az esetben a következő kódsorokat kell használnunk a megfelelő helyeken:

```
QSqlRecord *rec = cur.primeDelete();
// Csak lekérjük
cout << "\nRekordszám: " << cur.del();
```

A beszúrás műveletet a következő kódminta szemlélteti:

```
QSqlRecord *rec = cur.primeInsert();
// Csak lekérjük
rec->setValue("honap", 11);
rec->setValue("nap", 5);
rec->setValue("nevnap", "Imre" );
cout << "\nRekordszám: " << cur.insert();
```

Hibakezelés

Az SQL alapú hibák természetesen egy-egy adatbázis-kapcsolat viszonylatában merülnek fel, így a hibás állapotot is egy `QSqlDatabase` objektumon keresztül tudjuk lekérdezni, amire a `QSqlError` típusú objektumot visszaadó `lastError()` tagfüggvény szolgál. Például a következő kódsor a képernyőre az utolsó SQL-műveletben felmerült hiba típusát (a `pg` az adatbázis-kapcsolatot megtestesítő objektum) írja ki:

```
cout << "\nA hiba = " <<
➥(pg->lastError()).type();
```

Adatbázis-kezelő grafikus programok készítése

Eddig kizárólag olyan programokkal foglalkoztunk, amelyekben csak a szöveges képernyőt és a Qt SQL-modul illesztő- és adatkezelő részeit használtuk. Most áttekintjük a Qt SQL rendszer harmadik nagy részterületét, az adat(bázis)függő elemek (widgets) használatát és a grafikus programok készítésének a módját. Most csak egyszerű példákat mutatunk be, nem feltételezzük, hogy az olvasó ismeri a Qt grafikakészítés módszereit: jel-, illetve foglalat- (signal/slot) szerkezeteket, akciókat, eseményeket és a grafikus vezérlőket.
Nézzünk meg egy kicsi, de teljes mintapéldát, ami a nevek táblánk táblázatszerű kezelését mutatja meg (lásd még: 55. CD Magazin/Qt/tablazat_1.cpp), :

```
1. //
2. // tablazat_1.cpp
3. //
4. #include <iostream>
5. #include <qapplication.h>
6. #include <qsqldatabase.h>
7. #include <qsqlcursor.h>
8. #include <qdatatable.h>
9.
10.using std::cout;
11.
12.int main(int argc, char **argv)
13.{
14. QApplication app(argc, argv, true);
15. QSqlDatabase *pg =
        ➥QSqlDatabase::addDatabase
        ➥( "QPSQL7", "PG_CS_ADATOK" );
16. if ( !pg ) { cout << "Hiba a driver
        ➥betöltésekor!"; return (1); }
17.
18. pg->setDatabaseName("cs_adatok");
19. pg->setUserName("postgres");
20. pg->setPassword("111111");
21. pg->setHostName("localhost");
22. pg->setPort( 5432 );
23.
24. if ( !pg->open() )
25. { cout << "Hiba: az adatbázis nem
              ➥nyitható meg!";
26. return (1);
27. }
28.
29. QSqlCursor kurzor("ujnevek", true, pg);
30. kurzor.select("");
31.
32. QDataTable *t = new QDataTable(
        ➥&kurzor, false );
33.
34. t->addColumn("honap", "Hónap");
35. t->addColumn("nap", "Nap");
36. t->addColumn("nevnap", "Névnap");
37.
38. app.setMainWidget( t );
39.
40. t->refresh();
41. t->show();
42.
43. return app.exec();
44.}
```

Előzetesen érdemes megjegyezni, hogy a Qt-elemek mindegyike tároló- (container) és vezérlőelem egyaránt lehet, ami a veterán Windows-programozók számára talán még meglepő is. Bármelyik elem betöltheti a fő vezérlő szerepét is. A `tablazat_1.cpp` programban a `QDataTable` összetevőt alkalmaztuk, egyben az az alkalmazás fő vezérlője is. A 14. sorban meghatározott `app` alkalmazás az objektum harmadik értéke `true`, ami lehetővé teszi a grafikus elemek használatát. Ez újdonság az eddigiekhez képest. A 29–30. sorokban hoztuk létre azt a `kurzor` objektumot, ami egyben a grafikus tábla vezérlő adatforrása is lesz. A 32. sorban hozzuk létre a `t` mutatót, ami egy `QDataTable` objektumra mutat. A létrehozó első értéke a `kurzor` objektumunk címe. A második, `false` értékű azt jelenti, hogy mi akarjuk meghatározni a táblázat oszlopait. Amennyiben itt `true` értéket adunk, az oszlopok önműködően a sormutató oszlopaiból lesznek képezve (***auto populate*** üzemmód). Ebben az esetben a 34–36. sorra nem volna szükség. Az `addColumn()` tagfüggvények az oszlopokat határozzák meg, amelyekben az értékek jelentése ebben a sorrendben: egy sormutató oszlopváltozó, az oszlop fejléc címkéje. A 38. sorban táblázatunkat beállítjuk az alkalmazás fő vezérlőjének. A 40. sor `refresh()` tagfüggvénye felolvassa a kurzor sorait (hatékony, mert csak annyit, amennyi a megjelenítéshez szükséges) és ki is listázza azt. A 41. sor a vezérlőt megjeleníti a képernyőn.
A táblázatos kezelést megvalósító programunk futási képét az 55. CD Magazin/Qt könyvtárban lévő ***tablazat.jpg*** kép mutatja.

Űrlap alapú programok készítése

A Qt rendszerben a Delphi/C++ Builder-űrlapokhoz hasonló bonyolultságú űrlapokból álló alkalmazásokat is készíthetünk. Már most célszerű megemlíteni a Qt Designer nevű kiváló eszközt, amivel grafikusan készíthetjük el az űrlapokat, illetve az eseménykezelők és az események összekötését. A Qt Designer a Java nyelv elgondolásához hasonló – fejlett önműködő elemelrendezés-kezelővel (layout manager) rendelkezik. Az adatbázis alapú űrlapok készítésének áttekintéséhez nézzük meg a ***form1.cpp*** program forráskódját.

```
1. //
2. // form1.cpp
3. //
4. #include <qapplication.h>
5. #include <qdialog.h>
6. #include <qlabel.h>
7. #include <qlineedit.h>
8. #include <qsqlform.h>
9. #include <qsqldatabase.h>
```

```
10.#include <qsqlcursor.h>
11.#include <qdatatable.h>
12.#include <qlayout.h>
13.
14.////////////////////////////////
15.class TMyForm : public QDialog
16.{
17. public:
18. TMyForm();
19.
20. QSqlDatabase *pg;
21. QSqlCursor kurzor;
22.
23. QLineEdit *lNev;
24. QLabel *lHonap;
25. QLabel *lNap;
26. QGridLayout *grid;
27.};
28.
29.TMyForm::TMyForm()
30.{
31. pg = QSqlDatabase::addDatabase( "QPSQL7",
        ➥"PG_CS_ADATOK" );
32. // Kapcsolódás az adatbázishoz
33. pg->setDatabaseName("cs_adatok");
34. pg->setUserName("postgres");
35. pg->setPassword("111111");
36. pg->setHostName("localhost");
37. pg->setPort( 5432 );
38. pg->open();
39.
40. QSqlCursor kurzor("ujnevek", true, pg);
41. kurzor.select("honap='1'");
42. kurzor.next();
43.
44. lNev = new QLineEdit(this);
45. lHonap = new QLabel(this);
46. lNap = new QLabel(this);
47.
48. grid = new QGridLayout(this);
49. grid->addWidget( lNev, 0, 0 );
50. grid->addWidget( lHonap, 1, 0 );
51. grid->addWidget( lNap, 1, 1 );
52. grid->activate();
53.
54. QSqlForm aform(this);
55. aform.setRecord( kurzor.primeUpdate() );
56. aform.insert(lNev, "nevnap");
57. aform.insert(lHonap, "honap");
58. aform.insert(lNap, "nap");
59. aform.readFields();
60.}
61.
62.//--- A program indulási pontja ---
63.int main(int argc, char **argv)
64.{
65. QApplication app(argc, argv, true);
66.
67. TMyForm *f = new TMyForm();
68. f->show();
69. app.setMainWidget(f);
70. return app.exec();
71.}
```

A 15. sorban egy `TMyForm` osztályt vezetünk be, ami a `QDialog` utóda. Az osztály 15–27. sorában lévő megadásából (declaration) látható, hogy ugyanazokat a grafikus elemeket (címke, editor) használjuk, mint amikor nem használunk mögöttük adatforrásokat. A ***form1.cpp*** programban az osztályunk létrehozója minden lépést elvégez ahhoz, hogy az űrlapunk adatfüggő űrlap legyen. A 40–42. sor létrehozza az űrlap adatforrását. A 44–46. sor három új grafikus vezérlőt létesít, amiket a 48–52. sorban meghatározott `grid` nevű elrendezés-kezelő önműködően felpakol az űrlapunkra.
Nekünk csak azt kell megmondanunk, hogy a képzeletbeli rácscellák közül melyik vezérlőt hol (sor, oszloppozíció) szeretnénk látni. Végül az 54–59. sor az, ami űrlapunkból adatbázis-függő űrlapot varázsol. A feladatot egy `QSqlForm` típusú `aform` nevű objektum végzi el, aminek a tulajdonosa a `this`, azaz a mi későbbiekben létrehozandó `TMyForm` (a neve `(*f)` lesz és a `main()` függvényben látható) objektumunk lesz. Az 55. sor jegyzi be, hogy az adatforrás rekordjait honnan kell majd venni. Az 56–58. sor világosítja fel `aform` objektumunkat arról, hogy egy adott elemhez melyik sormutató-oszlopot szeretnénk társítani, majd az 59. sorban elvégezzük az elemek tényleges adatfeltöltését. A 63–71. sorban megírt `main()` főfüggvény működése során létrehozzuk a `TMyForm` objektumunkat, ami egyben a program fővezérlője is lesz.
A ***form1.cpp*** futtatható változatát az 55. CD Magazin/Qt könyvtárban lévő form.jpg képen láthatjuk.
Űrlapunk jelenleg csak annyit tud, hogy a sormutató első sorát megjeleníti egy űrlapos űrlapjában (Fruzsina, 1, 1 névnap, hó, nap). A navigálás, az adatmódosítás és az adatbázisba való mentés még nincs megoldva. A név mezőt már most is egy `QLineEdit` vezérlő tartalmazza, ezért a megjelenített első rekordnak ezt a mezőjét módosíthatjuk. De hogyan érvényesíthető ez a módosítás az adatbázisban? Erre szolgál a `QSqlForm::writeFields()` tagfüggvény. Az adatbázisban való módosítást tehát a következő kódrészlet mutatja be:

```
... változtatások a form vezérlői segítségével
aform.writeFields();
kurzor.update();
...
```

Ezt a kódrészt valakinek meg kell hívnia, azaz jellemzően egy eseménykezelőben kívánatos szerepeltetni. A `TMyForm` osztályt ki kell egészíteni egy `PUBLIC SLOT` szerepű, `mentes()` nevű tagfüggvénnyel, ami ezt az eseménykezelést valósítja meg. Ez a tagfüggvény így egészítené ki az osztályunkat:

```
class TMyForm : public QDialog
{
public:
TMyForm();
public slots:
void mentes();
...
}
```

A `public slots` címkét a `Qt moc` (Meta Object Compiler) program tünteti el, illetve szabványos C++-programot hoz létre belőle.
Ezek után feltehetünk egy `QPushButton` vezérlőt az űrlapra, amelynek a megvalósítása vázlatosan így nézne ki:

```
...
QPushButton *btnMentes = new
QPushButton("&Mentés", this);
...
grid->addWidget(btnMentes, 2, 2);
// Elhelyezzük az űrlapon
...
connect(btnMentes, SIGNAL(clicked()),
➥this, SLOT(mentes()));
...
```

A `btnMentes` gomb létrehozása után azt a már előzőleg létrehozott `grid` elrendezés-kezelő objektumunk segítségével feldobjuk az űrlapunkra. A `connect()` nagyon fontos függvény, mert ez köti össze az eseményt keltő objektum jelzését (signal) az eseményt kezelő objektum(ok) eseménykezelőivel, azaz a foglalattal. A `connect()` függvény értékeinek értelmezése:

1. A jelzést küldő objektum címe.
2. A jelzés (ez csak egy olyan tagfüggvényfej, amit soha nem kell megvalósítani, de meg kell határoznunk az értékeit, ha vannak; a `click()` jelzés érték nélküli).
3. Az eseményt kezelő objektum címe.
4. Az eseményt kezelő függvény, amit foglalatnak nevezünk (foglalat, mert ide csatlakoztathatjuk az eseménykezelést).

A ***form1.cpp*** programnak tehát azt a változatát is elkészíthetjük, amikor az adatbázisba mentő kód egy gomb megnyomása utá lefut.
A navigálás is hasonlóan oldható meg. Be kell vezetnünk a navigálógombokat (vagy más erre használható vezérlőt), amik mögé olyan eseménykezelőket (foglalatokat) kell írnunk, amik meghívják a `kurzor.next()...` tagfüggvényeket.

A Qt Designer használata

A gyakorlatban a komolyabb kiépítettségű űrlap alapú programjainkat általában nem soronként írjuk meg, így a Qt rendszerben is megtalálható a Qt Designer nevű grafikus fejlesztői környezet. A Designer az adatbázisfüggő űrlapok készítését is lehetővé teszi (*1. kép*).
Ez az eszköz a grafikai tervezés eredményét (űrlapok és eseménykezelők) egy ****.ui*** kiterjesztésű felhasználói kezelőfelület leíró XML-fájlban tárolja. Az ui fájlokból valódi C++-forrásprogram hozható létre, amelynek elkészítési folyamatát ábránk mutatja. A létrehozott C++-forrásprogram az űrlap osztályát megvalósító kódot is tartalmazza, amit az alkalmazás más helyein természetesen kedvünk szerint használhatunk, de a létrehozott forrásfájlokat kézzel ne változtassuk meg, illetve szükség esetén használjuk az öröklés lehetőségét. A Qt Designer szorosan össze van építve a KDevelop fejlesztői környezettel is, a két eszköz együttes használata nagyon termékeny.
A Designerben látható fenti űrlap egy `QDataBrowser` osztály alapú űrlap, ami a már ismert `QDataTable` osztály űrlap alapú testvére. Ez az osztály fogja tartalmazni a már ismertetett `QSqlForm` objektumot, így nekünk nem kell egy külön `TMyForm` szerepű osztályt gyártanunk. A fenti űrlapot a Designer segítségével a következő lépésekben hoztuk létre:

1. Egy teszt nevű új projekt létrehozása.
2. A ***Tools/Database/DataBrowser*** menüpont mögötti varázsló futtatása (létrehozza az űrlapot).
3. Mentés után a következő fájlok találhatók a lemezen: ***teszt.pro*** (ez a projektfájl), ***ab_qt_form1.ui*** (a megtervezett űrlap egy XML formátumban tárolt leírása), ***ab_qt_form1.h***.

A ***main.cpp*** főprogramot a következő minta alapján lehet létrehozni, illetve a Designerben ezt a fájlt is hozzá kell adni a ***teszt.pro*** projektünkhöz.

```
1. #include <qapplication.h>
2. #include <qsqldatabase.h>
3. #include
    ➥"/home/inyiri/cpp/qt_sql/.ui/ab_qt_form1.h"
4.
5. int main( int argc, char ** argv )
6. {
7. QApplication a( argc, argv );
8.
9. QSqlDatabase *pg =
   ➥QSqlDatabase::addDatabase( "QPSQL7" );
10. pg->setDatabaseName("cs_adatok");
11. pg->setUserName("postgres");
12. pg->setPassword("111111");
13. pg->setHostName("localhost");
14. pg->open();
15.
16. Form1 *w = new Form1;
17. w->show();
18. a.connect( &a, SIGNAL( lastWindowClosed()
    ➥), &a, SLOT( quit() ) );
19. return a.exec();
20.}
```

A 9–14. sorból láthatjuk, hogy az adatbázishoz történő kapcsolódást nekünk kell beprogramozni. A 16. sorban található `Form1` osztály a Designerben lett meghatározva. Természetesen nincs semmilyen korlát arra vonatkozóan, hogy a projektünk hány űrlapot tartalmazhat.
Ebből a négy fájlból a következő lépésekkel állítható össze egy futtatható program:

1. `qmake teszt.pro`
   A Qt `qmake` segédprogramja a Designer-projektfájl alapján létrehoz egy szabványos `Makefile`-t. Ez az eljárás a felületfüggetlen programozást is lehetővé teszi.
2. `make`
   A cikk példáit és a fordítást támogató `Makefile`-okat a magazin 55. CD-jének Magazin/Qt könyvtárában találhatjuk meg.

Mindenkinek kellemes programozást kívánok!

Irodalomjegyzék

1. A Qt csomaghoz adott leírások
➲ http://doc.trolltech.com
➲ http://www.trolltech.com
2. Linuxvilág 2002. október és 2002. december: Qt-programozás
3. Joe Celko: SQL felsőfokon
Kiskapu Kiadó (ISBN: 963 9301 20 5)

Nyíri Imre (inyiri@mol.hu)
Jelenleg a MOL Rt.-nél dolgozik. Informatikai vállalkozásában az Internet, a Linux, valamint a Java-programozás gyakorlati hasznosításával foglalkozik. Örök szerelme a C++ maradt.