مقدمه‌ای بر

توسعه‌ی برنامه‌های چندسکویی

با استفاده از چارچوب Qt

گردآوری و تألیف: مهرداد مومنی

با تشکر از همکاری و پشتیبانی فرید احمدیان

این کتاب برای چِه و کِه نوشته شده است؟

کتابچه‌ای که پیش رو دارید به منظور آشنایی اولیه با چارچوب Qt و حل مشکلات اولیه برنامه‌نویسان آماده شده است. در این کتاب مستقیما سراغ Qt و استفاده از امکانات آن در برنامه‌های C++ رفته‌ایم. پس خواننده باید با حداقل‌های C++ مخصوصا با کلاسها و بحث وراثت آشنایی هرچند اندک داشته باشد.

این کتاب باید بصورت ترتیبی خوانده‌شود چرا که مطالب هر فصل ارتباطی هرچند اندک با مطالب فصول قبل از آن دارد.

در نوشتن کتاب هدف بر خواننده‌ی فعال بوده، پس سعی کنید همراه با مطالب، مثالها و نمونه‌ها و پیشنهادات را تمرین کنید.

این کتاب چگونه گردآوری شده است؟

البته که این کتاب تولید محتوای صرف نبوده و در خیلی از موارد، مطالب، مثالها و نمونه کدهایی که در منابع اشاره‌شده است، مورد استفاده قرار گرفته، و بیشتر هدف مد نظر بوده تا تولید محتوا. پس تا جایی که مثالهای انگلیسی و معتبر موجود بوده از آنها استفاده شده است.

درباره‌ی مولف

مهرداد مومنی از سال ۸۶ با چارچوب Qt و توسعه نرم‌افزار در محیط سیستم‌عامل گنو/لینوکس آشنا شد، یک سال بعد به تیم توسعه‌ی میزکار KDE پیوست و پروژه‌ی Choqok را بعنوان کلاینت میکروبلاگ KDE آغاز کرد و در توسعه نرم‌افزار Blogilo (کلاینت رسمی وبلاگ نویسی KDE) و کتابخانه KBlog مشارکت نمود، وی در توسعه‌ی نرم‌افزار دیکشنری چندزبانه MDic نیز فعال بوده است.

فهرست

فصل اول: راه و روش Qt

Qt («کیوت» خوانده می‌شود.) یک چارچوب توسعه‌ی برنامه‌ی چند سکویی و گرافیکی است که به شما این امکان را می‌دهد که برنامه خود را روی سیستم عاملهای لینوکس، ویندوز، مکینتاش و برندهای مختلفی از یونیکس اجرا نمایید. بخش زیادی از کیوت، بگونه ای طراحی شده است که یک ظاهر طبیعی و شبیه به برنامه‌های اصلی آن سیستم برای سیستم عاملهای مختلف ایجاد نماید، از نگهداری متون در حافظه گرفته تا ساختن یک برنامه‌ی گرافیکی چند نخی(thread).

برای شروع یادگرفتن انجام کارها به روش Qt در این بخش یک برنامه‌ی ساده C++ را در نظر گرفته و آنرا به روش کیوت بازنویسی می‌نماییم.

نصب و آماده سازی یک محیط توسعه برای Qt

از جایی که سیستم عاملهای ویندوز و گنو/لینوکس پرکاربردتر هستند، بحثهایی که بنا به محیط متغیر هستند را برای این دو محیط بسط خواهیم داد.

امروزه نصب محیط توسعه کیوت، در سیستمهای عمومی تر، ساده تر از همیشه هستند، چرا که دیگر نیازی به کامپایل آن از سورس نیست. هم برای محیط ویندوز و هم برای توزیع های مختلف گنو/لینوکس پکیجهای کامپایل شده و آماده‌ی نصب در دسترس کاربران می‌باشند.

نصب در محیط گنو/لینوکس

در محیط گنو/لینوکس شما ۳ گزینه برای نصب دارید، از بسته هایی که توزیع کننده‌ی لینوکس شما آماده کرده است استفاده نمایید، که ساده تر، سریعتر و مطمئن‌تر است، یا سورس کیوت و نیازمندی های آنرا دانلود نموده کامپایل، و نصب نمایید، و یا از بسته‌های عمومی که نوکیا (شرکت توزیع‌کننده‌ی کیوت) منتشر می‌کند استفاده نمایید.

برای استفاده از بسته های آماده، کافی است پکیج(بسته) ای که توزیع کننده‌ی لینوکس شما آماده کرده است را نصب نمایید، برای مثال در توزیعهای برپایه دبیان (مثل اوبونتو، مینت یا پارسیکس) بسته libqt4-dev را که نصب کنید، کلیه کتابخانه های کیوت و نیازمندیهای آنها نصب خواهند شد.

و سپس شما نیاز به نصب برنامه های کمکی کیوت (Qt Assistant) طراح کیوت(Qt Designer) و یک محیط مجتمع توسعه(IDE) دارید، که برای محیط توسعه می‌توانید از Qt Creator و یا KDevelop4 استفاده نمایید.

بعد از نصب این برنامه ها، و نصب کامپایلر GCC و دیباگر GDB محیط توسعه‌ی شما آماده استفاده خواهد بود.

نصب در محیط ویندوز

نصب در محیط ویندوز، با نصب کردن پکیجی که تحت عنوان Qt SDK منتشر می‌شود (این بسته برای گنو/لینوکس و مک هم در دسترس است.)، به سادگی امکان پذیر می‌باشد، کافی است این فایل را از سایت Qt.Nokia.com دانلود و نصب نمایید، بعد از نصب شما همه‌ی برنامه های ذکر شده برای لینوکس (غیر از KDevelop4) را خواهید داشت.

C++ را Qt ای کنیم!

شاید این جمله عجیب به نظر بیاید! اما حقیقت دارد، چارچوب توسعه‌ی کیوت افزونه‌های مفیدی برای زبان C++ دارد که تجربه‌ی شما از C++ را تغییر خواهد داد.

چون این کتاب در مورد برنامه نویسی است، با یک کد شروع می‌کنیم:

```
#include <string>

using std::string;

class MyClass
{

public:
      MyClass( const string& text );

      const string& text() const;
      void setText( const string& text );

      int getLengthOfText() const;

private:
      string m_text;
};
```

کلاسی که در کد بالا آورده شده است، یک کلاس ساده C++ می‌باشد.

یک رشته داریم، که می‌توان به آن دسترسی داشت، مقدار آنرا عوض کرد، و یا طول آنرا حساب کرد. اما ابتدا ببینیم کدام قسمتهای کد شبیه کیوت هست و نیازی به تغییر ندارد...

کدام قسمتهای این کد به کیوت شبیه است:

- اسم کلاس با کاراکتر بزرگ شروع شده است، و نامگذاری از روش Camel Casing تبعیت می‌کند. یعنی هر اسم جدیدی با کاراکتر بزرگ شروع می‌شود، این چیزی است که در نامگذاری کلاسها در کیوت رعایت می‌شود. (البته هیچ اجباری در استفاده از این روش نیست، در اینجا منظور از نامگذاری کلاسها در کیوت، در کتابخانه‌ی کیوت است، در حالی که شما در برنامه‌ی خود می‌توانید از هر اسلوب دیگری نیز پیروی کنید.)
- اسم توابع با حرف کوچک شروع می‌شود، اما در ادامه از همان روش Camel Casing پیروی می‌کند، یعنی کلمات بعدی در اسامی با حرف بزرگ شروع می‌شوند.
- متدهای getter(گیرنده‌ی مقدار) و setter(تنظیم کننده مقدار) برای خصوصیت(Property) text بصورت text() و setText() نامگذاری شده اند. این روش عمومی نامگذاری آنها در کیوت می‌باشد. شما هم می‌توانید از آن پیروی کنید، یا نکنید.

این مسائل ممکن است چندان به چشم نیاید، اما نامگذاری به یک روش خاص، زمانی که شما برنامه‌های بزرگ‌تری می نویسید، خیلی مفید و کاربردی خواهد بود.

ارث‌بری از Qt

اولین تغییری که برای کیوت‌ای کردن کد انجام می‌دهیم: ارث‌بری کردن کلاسهایمان از کلاس QObject است. که باعث مدیریت ساده تر و بهتر اشیاء می‌گردد.

تغییرات در کلاس ساده خواهند بود، اولین مورد به شرح زیر است:

```
#include <QObject>
#include <string>

using std::string;

class MyClass : public QObject
{
public:
      MyClass( const string& text, QObject *parent = 0 );
...
};
```

توجه: برای دسترسی به کلاس QObject لازم است که هدرفایل QObject را include نماییم. این سیستم برای اکثر کلاسهای کیوت کار خواهد کرد، یعنی برای استفاده از آنها، فایلی هم نام با کلاس را include کنید.

پارامتر parent همانگونه که هست، به سازنده‌ی کلاس مادر، یعنی QObject پاس داده می‌شود:

```
MyClass::MyClass( const string& text, QObject *parent ) : QObject( parent )
```

حالا به تغییری که این کار در نحوه‌ی کد زدن ما ایجاد می‌کند نگاهی می‌اندازیم، ابتدا نحوه استفاده از کلاس MyClass در C++ بدون حضور Qt:

```
#include <iostream>
int main( int argc, char **argv )
{
      MyClass *a, *b, *c;
      a = new MyClass( "foo" );
      b = new MyClass( "ba-a-ar" );
      c = new MyClass( "baz" );
      std::cout << a->text() << " (" << a->getLengthOfText() << ")" <<
      std::endl;
      a->setText( b->text() );
      std::cout << a->text() << " (" << a->getLengthOfText() << ")" <<
      std::endl;
      int result = a->getLengthOfText() - c->getLengthOfText();
      delete a;
      delete b;
      delete c;
      return result;
}
```

برای جلوگیری از نشت حافظه(memory leak) هرکدام از فراخوانی های new باید با یک فراخوانی delete همراه باشد. البته در اکثر سیستمهای مدرن که با خروج برنامه سیستم عامل فضاها را آزاد می‌کند، ممکن است مشکل بوجود نیاورد، اما همین نشت حافظه می‌تواند باعث کرش کردن برنامه در شرایطی که سیستم با کمبود حافظه مواجه است گردد. و هرچه برنامه بزرگتر می‌شود، این خطاها بیشتر باعث دردسر آن می‌گردند.

حال بیایید این کد را با کد بعدی که با استفاده از یک والد که بطور خودکار با به پایان رسیدن تابع main حذف می‌شود و خودش مسئول آزاد کردن حافظه‌ی فرزندانش خواهد بود، مقایسه کنیم:

توجه: در کدی که در ادامه می‌آید، شیئ parent صرفا برای نشان دادن این مفهوم به برنامه افزوده شده است، در حالی که در برنامه های واقعی همیشه حداقل یک شیئ برای اینکار داریم، مثلا یک شیئ QApplication و یا پنجره‌ی برنامه.

```
#include <QtDebug>
int main( int argc, char **argv )
{
QObject parent;
MyClass *a, *b, *c;
a = new MyClass( "foo", &parent );
b = new MyClass( "ba-a-ar", &parent );
c = new MyClass( "baz", &parent );
qDebug() << QString::fromStdString(a->text())
              << " (" << a->getLengthOfText() << ")";
a->setText( b->text() );
qDebug() << QString::fromStdString(a->text())
              << " (" << a->getLengthOfText() << ")";
return a->getLengthOfText() - c->getLengthOfText();
}
```

شما حتی قسمتی از برنامه که برای نگهداری نتیجه‌ی تفریق نهایی بود را نیز حذف کردید، چرا که اشیائی که بطور اتوماتیک ایجاد می‌شوند، میتوانند بعنوان یک پارامتر به دستور return پاس داده شوند.

ایجاد یک شیئ بعنوان والد(parent) ممکن است زشت به نظر برسد اما اکثر برنامه های کیوت، یک شیئ از نوع QApplication دارند که بدین منظور میتوان از آن استفاده نمود.

توجه: در کد آخری که داشتیم، دیدید که بجای استفاده از std::cout از تابع qDebug استفاده کردیم، برتری qDebug این است که پیغام را در همه‌ی سیستمها به همانجا که لازم است می‌فرستد. همچنین به سادگی میتوان آنرا غیرفعال نمود، کافی است در زمان کامپایل QT_NO_DEBUG_OUTPUT را تعریف کنید. Qt همچنین ۲ تابع دیگر برای این منظور آماده کرده است، qFatal که بعد از نشان دادن پیغام برنامه را نیز می‌بندد، در زمانهایی که خطای مهلکی رخ داده و برنامه نمیتواند ادامه پیدا کند قابل استفاده است. و qWarning که نشان می‌دهد پیغام مورد نظر یک خطا است (به هر حال نشان داده شود) اما خطای مهلکی نیست که برنامه را ببندد. همچنین همه‌ی این توابع خروجی خود را بهمراه یک std::endl به پایان می‌برند. و دیگر نیازی به استفاده از آن نیست.

در عکس زیر می‌توانید تفاوت بین دو حالت استفاده از یک شیئ که در پشته (stack) قرار دارد بعنوان والد و عدم استفاده از آنرا ببینید.

والد بصورت خاکستری نشان داده شده است، چرا که در حافظه‌ی stack قرار دارد، و بطور اتوماتیک حذف می‌شود(حذف شدن اشیاءی که در stack هستند جزء طبیعت C++ است.). و فرزندان خود را حذف خواهد کرد(حذف کردن فرزندان جزء طبیعت QObject و کلاسهای وارث آن(همه‌ی کلاسهای Qt) است).

می بینیم که به همین سادگی یک قابلیت بسیار مفید که C++ فاقد آن می‌باشد و در اصطلاحات فنی به آن Garbage Collector می‌گویند. و زبانهایی مثل Java با داشتن آن به C++ فخر می‌فروشند را به C++ اضافه نمودیم.

استفاده از یک رشته‌ی Qt

قدم بعدی در راه استفاده از کیوت، جایگزین کردن هر کلاس استاندارد C++ با کلاس معادل آن در کیوت می‌باشد(البته در صورت وجود کلاسی معادل).

توجه: ممکن است در وهله اول این سخن برایتان سخت بیاید، اینکه کیوت برای بیشتر کلاسها و روشهای استاندارد C++ یک کلاس معادل دارد، و ممکن است شبهه انحصارطلبی در ذهن ایجاد کند، اما در ادامه‌ی زندگی با کیوت، درک خواهید کرد که اگر کلاسی، معادلی در کیوت دارد، صرفا به دلیل وجود کمبودهایی در کلاس استاندارد می‌باشد. و هیچگونه عمدی به انحصارطلبی در این زمینه در کار نبوده است. همچنین خواهید دید که کیوت بنحو خیلی خوبی با کتابخانه‌ی استاندارد C++ (std) کار می‌کند. و هیچگونه مشکلی در استفاده از آنها بهمراه کیوت نخواهید داشت.

یکی از مهمترین کلاسها و زمینه‌های کاری که C++ در آن ضعیف است، کلاس string و کار با رشته ها است. کیوت با توسعه کلاس QString این ضعف C++ را تا حدودی جبران نموده است.

یکی از مهمترین خصوصیات و برتریهای QString مخصوصا برای دنیای امروز و ما فارسی زبانان، این است که متون در آن بطور پیشفرض بصورت Unicode ذخیره می‌گردند. پس هیچگونه مشکلی با متون فارسی نخواهیم داشت.

در کدی که در ادامه می‌آید، کد کلاس MyClass را تغییر داده و از QString بجای std::string استفاده نموده‌ایم.

```
#include <QString>
#include <QObject>
class MyClass : public QObject
{
public:
      MyClass( const QString& text, QObject *parent = 0 );
      const QString& text() const;
      void setText( const QString& text );
      int getLengthOfText() const;
private:
      QString m_text;
};
```

زمانی که با هر دو کلاس std::string و QString کار می‌کنید، از توابع toStdString و fromStdString برای تبدیل رشته به string و گرفتن رشته از string استفاده نمایید.

کامپایل برنامه

کامپایل کردن برنامه‌ی نهایی هیچ تفاوتی با کامپایل کردن کد اولی نخواهد داشت، تنها باید مطمئن شوید که کامپایلر هدرفایلهای(header files) کیوت را پیدا خواهد کرد، و همچنین برنامه لینکر(linker) کتابخانه های کیوت را پیدا خواهد کرد.

برای انجام همه این کارها بصورت ساده و سریع، یک ابزار مفید بنام QMake بهمراه کیوت منتشر می‌گردد. که می‌تواند Makefile (فایلی شامل دستوراتی که نحوه‌ی کامپایل کردن کد را برای برنامه‌ی GNU Make مشخص می‌کند.) برای یک بازه‌ی وسیعی از کامپایلرها تولید کند. (در صورت آشنایی با ابزارهای دیگری مثل AutoMake و یا CMake می‌توانید از آن‌ها بجای QMake استفاده نمایید.)

اینکار را با ساختن یک برنامه ساده شروع می‌کنیم، با ساختن یک پوشه بنام project آغاز کنید، سپس کدی که در زیر می‌آید را در فایلی بنام main.cpp در این پوشه ذخیره نمایید.

```
#include <QtDebug>
int main( )
{
      qDebug() << "Hello Qt World!";
      return 0;
}
```

حال در خط فرمان به پوشه‌ی project بروید. تایپ کنید: qmake –project و کلید Enter را فشار دهید. اگر یک لیست از محتوای پوشه بگیرید، مشاهده خواهید نمود که فایلی بنام project.pro در آن ایجاد شده است.

نکته: اگر نسخه آزاد کیوت برای ویندوز را نصب کرده باشید، یک برنامه بنام Qt Command Prompt همراه آن نصب شده است، که برای این منظور می‌توانید از آن استفاده نمایید.

اگر فایل project.pro را باز کنید، احتمال چیزی شبیه به این خواهید داشت:

```
#############################################################
# Automatically generated by qmake (2.00a) to 10. aug 17:06:34 2006
#############################################################
TEMPLATE = app
TARGET +=
DEPENDPATH += .
INCLUDEPATH += .
# Input
SOURCES += main.cpp
```

که یک سری متغیر را با عملگر = مقدار دهی کرده است، و یا مقداری به یکسری متغیر با عملگر += افزوده است.

بخش جالب آن متغیر SOURCES می‌باشد که نشان می‌دهد qmake فایل main.cpp که سورس ما می‌باشد را یافته است. (در آینده با دیگر بخشهای مهم این فایل آشنا خواهید شد.)

قدم بعدی ساختن یک Makefile وابسته به سیستم از این فایل است، این مهم با اجرای دستور زیر عملی می‌گردد:

```
qmake
```

و یا با اجرای:

```
qmake project.pro
```

چون در پوشه‌ی کنونی تنها یک فایل پروژه(pro.) موجود است، با اجرا qmake خودبخود این فایل انتخاب می‌گردد، اما اگر چندین فایل .pro وجود داشت، نیاز به متمایز کردن آن داشتیم.

نتیجه‌ی اجرای این دستور، یک Makefile و فایلهای دیگری که برای کامپایل برنامه مورد نیاز هستند است.

قدم آخر کامپایل برنامه با اجرای دستور make (در ویندوز: mingw32-make) در پوشه‌ی مورد بحث است.

آخرین بخش باقیمانده هم اجرای برنامه می‌باشد. که در لینوکس با اجرای دستور project/. و در ویندوز با اجرای project.exe عملی می‌باشد.

نکته: در صورتی که بخواهید در ویندوز خروجی در خط فرمان داشته باشید، باید مقدار console را به متغیر CONFIG بیافزایید، مانند زیر:

```
CONFIG += console
```

بررسی یک برنامه با واسط دیداری که با Qt نوشته شده است:

در ادامه مبحث روش کیوت، یک برنامه با واسط گرافیکی را بررسی می‌کنیم. کد زیر را ببینید:

```
1. #include <QApplication>
2. #include <QLabel>
3. int main(int argc, char *argv[])
4. {
5.   QApplication app(argc, argv);
6.   QLabel *label = new QLabel("Hello Qt!");
7.   label->show();
8.   return app.exec();
9. }
```

تابع main در یک برنامه‌ی استاندارد کیوت، شامل تعریف یک متغیر از نوع QApplication می‌باشد، که منابعی در سطح برنامه(application) را مدیریت می‌کند. سازنده آن argc و argv که ورودیهای خط فرمان برنامه هستند، را قبول می‌کند.

خط ۶ یک شیئ از ویدجتِ (widget) برچسب (QLabel) تعریف می‌کند.

نکته: در اصطلاحات یونیکس و کیوت، کلمه ویدجت (widget که d (دال) آن خوانده نمیشود) به عنصری که نمود دیداری دارد اطلاق می‌گردد. ریشه‌ی این کلمه gadget window است. که معادل کنترل(control) یا نگهدارنده(container) در اصطلاحات ویندوز است. دکمه ها(buttons) منوها(menus) و فریم ها(frames) همه مثالهایی برای ویدجت هستند. ویدجتها می‌توانند دیگر ویدجتها را نگهدارند. برای مثال، پنجره اصلی یک برنامه یک ویدجت است، که دارای چند QStatusBar، QToolbar، QMenuBar و ویدجتهای دیگری است. اکثر برنامه ها از QMainWindow و یا QDialog برای پنجره اصلی برنامه استفاده می‌کنند. اما کیوت آنقدر انعطاف پذیر است که هر ویدجتی می‌تواند یک پنجره باشد. در این مثال برچسب ما خود، پنجره برنامه است.

خط ۷ برچسب را نمایان می‌کند. ویدجتها بطور پیش فرض بصورت مخفی ساخته می‌شوند. تا بتوانیم قبل از نمایش، آنها را تنظیم نماییم.

شاید برای شما این سؤال پیش آمده باشد که چرا شیئ برچسب را حذف(delete) نکردیم، در حالی که بعنوان فرزند هیچ ویدجتی هم تعریفش نکردیم! کاملاً درست است، اما در این حد کم و کوچک، زیاد مهم نیست. و از جایی که اینجا سیستم‌عامل بطور خودکار با بسته شدن برنامه حافظه را آزاد می‌کند، اهمیت زیادی ندارد. اما بطور کلی سعی کنید همیشه اشیاء Qt را در زمان اجرا یا بعد از آن به قسمتی از درخت اشیاء بچسبانید.

هیچ عکسی از نتیجه‌ی اجرا نمی‌آوریم، تا خودتان آنرا آزمایش کنید! با توجه و استفاده از عملیاتی که برای کامپایل و تست کد در قسمت قبل توضیح دادیم، این کد را کامپایل و اجرا نمایید.

قبل از به اتمام رساندن بحث این برنامه، بیایید کمی تفریح کنیم

در برنامه‌ای که اجرا کردید، خط زیر را

```
QLabel *label = new QLabel("Hello Qt!");
```

با

```
QLabel *label = new QLabel("<h2><i>Hello</i> "
                           "<font color=red>Qt!</font></h2>");
```

تعویض نمایید! و برنامه را دوباره کامپایل و اجرا نمایید. (برای کامپایل دوباره‌ی برنامه، اجرای دستور make بتنهایی کفایت می‌کند.)

همانگونه که می‌بینید، اکثر ویدجتهایی که متنی نمایش می‌دهند(مانند Qlabel و QTextEdit)، متن با فرمت html نیز قبول می‌کنند، و این مسئله دست برنامه‌ساز را در تعریف واسط برنامه باز می‌گذارد.

نکته: البته توجه داشته باشید که Qt ورودی html را بطور کامل پشتیبانی نمی‌کند! و در این زمینه کاستی‌هایی دارد. برای اطلاع از بازه‌ی تگهای html ای که پشتیبانی می‌شوند سری به آدرس زیر بزنید.

http://doc.qt.nokia.com/latest/richtext-html-subset.html

مدل اشیاء در Qt

آخرین بحث ما در این فصل یک جمع بندی در مورد مدل اشیاء در Qt است.

مدل اشیاء C++ به نحو بهینه ای از مدل اشیاء در زمان اجرا پشتیبانی می‌کند. اما طبیعت ایستای آن(static) در بعضی از زمینه ها غیرقابل انعطاف می‌باشد.

برنامه نویسی واسط کاربری یکی از زمینه های برنامه نویسی است که هم به قابلیت های زمان اجرا نیاز دارد و هم به انعطاف پذیری. کیوت این مقوله را با ترکیب نمودن سرعت C++ و انعطاف پذیری بالای Qt به ارمغان می‌آورد.

Qt امکانات زیر را به C++ می‌افزاید و کمبودهای آنرا جبران می‌نماید:

- یك مکانیزم قدرتمند برای پیاده سازی ارتباطات یکپارچه بین اشیاء که سیگنال‌ها و اسلات‌ها(signals and slots) نامیده می‌شود.
- خصوصیات(properties) برای اشیاء که قابل جستجو و طراحی می‌باشد.
- رخداد(event)های قدرتمند و فیلتر کردن آنها.
- قابلیت ترجمه شدن عبارات و جملات برنامه بدون تغییر کد که ترجمه و بین المللی سازی برنامه‌ها را ساده می‌سازد.
- تایمرهای قدرتمند(کلاس QTimer)
- سلسله مراتب قابل درخواست درخت اشیاء که مالکیت اشیاء را در سیستم مدیریت می‌کند.
- اشاره‌گرهای محافظت شده (QPointer) که بصورت اتوماتیك مقدارشان در زمانی که به هیچ شیئی اشاره نمی‌کنند صفر می‌گردد.
- تبدیل انواع داده‌ای پویا(Dynamic Casting) که در محدوده‌ی اشیاء Qt کار می‌کند.

بسیاری از این قابلیت های کیوت بوسیله‌ی تکنیکهای استاندارد C++ پیاده‌سازی شده‌اند. مثلا برپایه ارث‌بری از QObject.

دیگر قابلیتها همچون مکانیزم ارتباطی اشیاء و سیستم پویای خصوصیتها نیاز به سیستم Meta Object ای که به همراه کامپایلر متا آبجکت(Meta Object Compiler) کیوت منتشر می‌گردد، دارد.

سیستم متا آبجکت یك افزونه برای C++ است که این زبان را برای پشتیبانی بهتر از برنامه نویسی کامپوننتی آماده می‌سازد.

کلاسهای مهم:

QObject	کلاس پایه برای همه‌ی اشیاء Qt (بعضی کلاسها مثل QString یک شیء نمی‌سازند بلکه تنها یک نوع داده محسوب می‌شوند.)
QPointer	کلاس الگو(template) برای ارائه دادن اشاره‌گرهای محافظت شده.
QVariant	بعنوان یک اجتماع برای نوع داده‌های Qt عمل می‌کند.

فصل دوم: سیگنال‌ها و اسلات‌ها در Qt

سیگنال‌ها و اسلات‌ها یکی دیگر از امکاناتی است که Qt به برنامه‌نویس C++ می‌دهد، البته این سیستم بعد از کیوت در بعضی دیگر از ابزارها نیز پیاده‌سازی شد. این مکانیزم به برنامه‌نویس امکان ارتباط ساده و سریع بین اشیاء برنامه را می‌دهد.

سیگنال‌ها توابعی هستند که ساتع می‌گردند(emit)، نه اینکه در زمان فراخوانی(call) اجرا(execute) گردند. پس از دید ما، بعنوان برنامه‌نویس، ما پروتوتایپ(prototype) تابع را تعریف می‌کنیم! ولی آنرا پیاده‌سازی نمی‌کنیم.

یک اسلات هم یک تابع است که می‌تواند احضار گردد در نتیجه‌ی ساتع شدن یک سیگنال.

بگذارید بحث را با یک مثال دیداری ادامه‌دهیم. کد زیر را ببینید:

```
 1 #include <QApplication>
 2 #include <QPushButton>
 3 int main(int argc, char *argv[])
 4 {
 5     QApplication app(argc, argv);
 6     QPushButton *button = new QPushButton("Quit");
 7     QObject::connect(button, SIGNAL(clicked()),
 8                      &app, SLOT(quit()));
 9     button->show();
10      return app.exec();
11 }
```

یک برنامه‌ی ساده Qt است که پنجره‌ی اصلی آن یک دکمه با متن Quit است. برنامه را کامپایل و اجرا کنید، و بعد از اجرا روی دکمه‌ی Quit کلیک کنید، همانطور که انتظار می‌رفت، برنامه بسته می‌شود. چرا؟

خط ۷ برنامه سیگنال کلیک‌شدن دکمه را به اسلات خروج(QUIT) برنامه متصل کرده‌ایم! پس هرگاه این سیگنال ساتع گشت، برنامه بسته خواهدشد.

اشیاء کیوت، سیگنال‌هایی برای سادگی کار با آن‌ها دارند، که با مراجعه به مستندات هرکدام می‌توانید از آن‌ها مطلع گردید.

تابع connect در کلاس QObject تعریف شده است، پس هر فرزند (subclass) از آن این تابع را خواهد داشت.

می‌توان هر سیگنال را به هر تعداد اسلات متصل کرد، و مطمئن بود در زمان ساتع شدن آن سیگنال همه‌ی آن اسلاتها اجرا خواهند شد. و حتی بیشتر، می‌توان یک سیگنال را به یک سیگنال دیگر متصل کرد، و اینگونه با ساتع شدن سیگنال اول، دومی هم ساتع خواهد شد.

برای بازتر کردن بحث، به کد کلاس MyClass که در بخش قبل نوشتیم برمی‌گردیم...
کد زیر تغییر یافته‌ی کلاس ما است:

```
#include <QString>
#include <QObject>

class MyClass : public QObject
{
    Q_OBJECT
public:
    MyClass( const QString &text, QObject *parent = 0 );
    const QString& text() const;
    int getLengthOfText() const;

public slots:
    void setText( const QString &text );

signals:
    void textChanged( const QString& );

private:
    QString m_text;
};
```

در سه بخش تغییراتی در سورس ما بوجود آمده است، که آن‌ها را از پایین به بالا بررسی می‌کنیم:

در پایین تعریف کلاس، بخشی بنام signals آمده است، همانطور که حدس زده‌اید، در این بخش ما سیگنال‌های کلاسمان را تعریف می‌کنیم، هر چند تا سیگنالی که لازم داشته باشیم. دوباره تأکید می‌کنم، نباید سیگنال را پیاده‌سازی کنید! یعنی کد سیگنال به همین یک خط الگوی تابع تمام می‌شود. (البته از دید برنامه‌نویس، اما از دید فنی، پیاده‌سازی این تابع را کامپایلر متا آبجکت کیوت، آماده خواهد کرد.)

بالاتر می‌رویم، تابع setText که از قبل داشتیم را در قسمت اسلات‌های عمومی تعریف کرده ایم.

اسلات‌ها می‌توانند عمومی(public) محافظت‌شده(protected) و یا خصوصی(private) تعریف گردند. اما سیگنال‌ها همیشه عمومی هستند. اسلاتها را می‌توان بعنوان توابعی از کلاس که می‌توانند به یک سیگنال متصل گردند در نظر گرفت، صرفاً تفاوت خاص دیگری نیز ندارند، پس به آن‌ها به چشم یک تابع نگاه کنید.

در بالاترین قسمت تعریف کلاس، شما ماکروی Q_OBJECT را می‌بینید. این عبارت باید در ابتدای تعریف کلاس باشد. تا این کلاس را به عنوان یکی از کلاسهایی که باید برای آنها meta-object ساخته‌شود، مشخص کنیم. از دید Qt متا آبجکتها نمونه‌هایی از کلاس QMetaObject هستند که اطلاعاتی در مورد کلاس مربوطه از جمله نام کلاس، کلاس پایه‌ی آن، سیگنال‌ها و اسلات‌های آن و خیلی اطلاعات جذاب دیگر دارند.

تا به حال برنامه‌هایی که نوشتیم و اجرا کردیم، در یک فایل تعریف شده بودند، اما خیلی زیباتر می‌شود وقتی که هر کلاس را در دو فایل header و source که با پسوندهای h و cpp مشخص می‌شوند تقسیم کنیم. یک ابزار کیوت که قبلاً هم از آن نام بردیم، کامپایلر متا آبجکت (moc)، همه‌ی کلاسها را بررسی می‌کند، و در صورت نیاز یک فایل پیاده‌سازی متا آبجکت برای هرکدام ایجاد می‌کند (فایلی با نام moc_FILENAME.cpp). این به نظر پیچیده می‌آید، اما از جایی که شما از QMake استفاده می‌کنید، اهمیتی ندارد، چون از پس این مسئولیت نیز به خوبی برمی‌آید.

این می‌طلبد که سورس بالا را در یک فایل بنام myclass.h ذخیره کنیم، و کد پیاده‌سازی کلاس را در فایل myclass.cpp که moc یک فایل دیگر بنام moc_myclass.cpp ایجاد خواهد کرد.

سورس زیر تغییراتی که در پیاده‌سازی کلاس MyClass ایجاد می‌گردد را نشان می‌دهد:

```
1 void MyClass::setText( const QString &text )
2 {
3   if( m_text == text )
4         return;
5   m_text = text;
6   emit textChanged( m_text );
7 }
```

در ابتدای تابع چک می‌کنیم که آیا واقعاً متن تغییری کرده است!؟ تا اگر نکرده باشد، سیگنال textChanged را ساتع نکنیم.

ساتع کردن یک سیگنال به سادگی دستور خط ۶ است:

```
emit SIGNAL_NAME (  PARAMETERS  );
```

یا:

```
Q_EMIT SIGNAL_NAME (  PARAMETERS  );
```

هر دو ماکرو تعریف شده هستند.

نکته: سیگنال‌ها و اسلاتها در کیوت با استفاده از اشاره‌گر تابع پیاده‌سازی شده‌اند. پس زمانی که دستور emit را بهمراه سیگنال فراخوانی می‌کنیم، درواقع تابع سیگنال را فراخوانی نموده‌ایم. همانطور که پیشتر دیدیم، برای هر سیگنال یک پیاده‌سازی در فایلی که moc ایجاد کرده است، وجود دارد. و در آن فراخوانی سیگنال همه‌ی اسلات‌های متصل به آن را بهمراه آرگومانها فراخوانی می‌کند. در این زمان برنامه، ورودی اسلات را چک می‌کند، و صرفاً آرگومانهایی را که اسلات می‌خواهد به آن پاس می‌دهد، و از دیگر آرگومانها صرف نظر می‌کند. این چک کردن در سمت سیگنال انجام نمی‌پذیرد، یعنی اسلات می‌تواند همه یا بخشی از آرگومان‌های سیگنال را بعنوان ورودی بگیرد. و این یعنی یک اسلات که ورودی ندارد، می‌تواند با هر سیگنالی هماهنگ گردد. *اما اگر ورودی اسلات آرگومانی دارد که در سیگنال نیست، یا به همان ترتیب نیست، اسلات فراخوانی نخواهد شد.* پس در هماهنگ‌سازی سیگنال‌ها و اسلات‌ها دقت کنید.

نحوه‌ی استفاده و عمل‌کرد سیگنال و اسلات تعریف شده در کلاس را بررسی می‌کنیم:

ابتدا سه متغیر تعریف می‌کنیم:

```
QObject parent;
MyClass *a, *b, *c;
a = new MyClass( "foo", &parent );
b = new MyClass( "bar", &parent );
c = new MyClass( "baz", &parent );
```

کد به نظر آشنا می‌رسد نه!؟

حال ارتباط بین اشیاء را تعریف می‌کنیم:

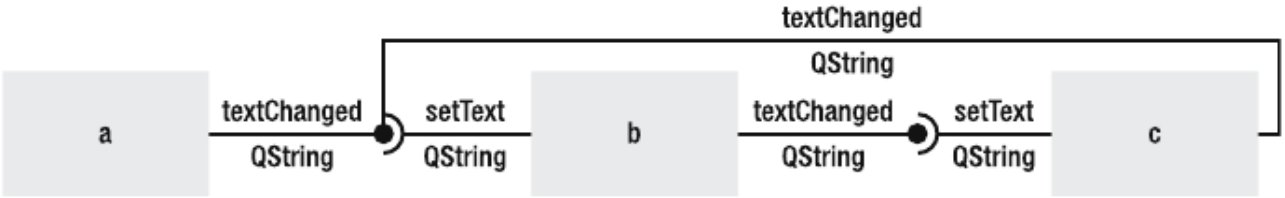
```
QObject::connect(a, SIGNAL(textChanged(const QString&)),
                 b, SLOT(setText(const QString&)) );
QObject::connect(b, SIGNAL(textChanged(const QString&)),
                 c, SLOT(setText(const QString&)) );
QObject::connect(c, SIGNAL(textChanged(const QString&)),
                 b, SLOT(setText(const QString&)) );
```

ورودی تابع connect بصورت زیر است:

```
connect( SenderObject, SIGNAL,
          ReceiverObject, SIGNAL_or_SLOT)
```

نکته: تلاش برای مقداردهی به سیگنال یا اسلات در زمان برقراری ارتباط(اجرای connect) باعث مشکل در زمان اجرای برنامه خواهد شد. صرفاً نوع متغیر ورودی سیگنال و اسلات در تابع connect لازم است.

ارتباط بین سه شیئ a , b و c را در شکل زیر می‌بینید:

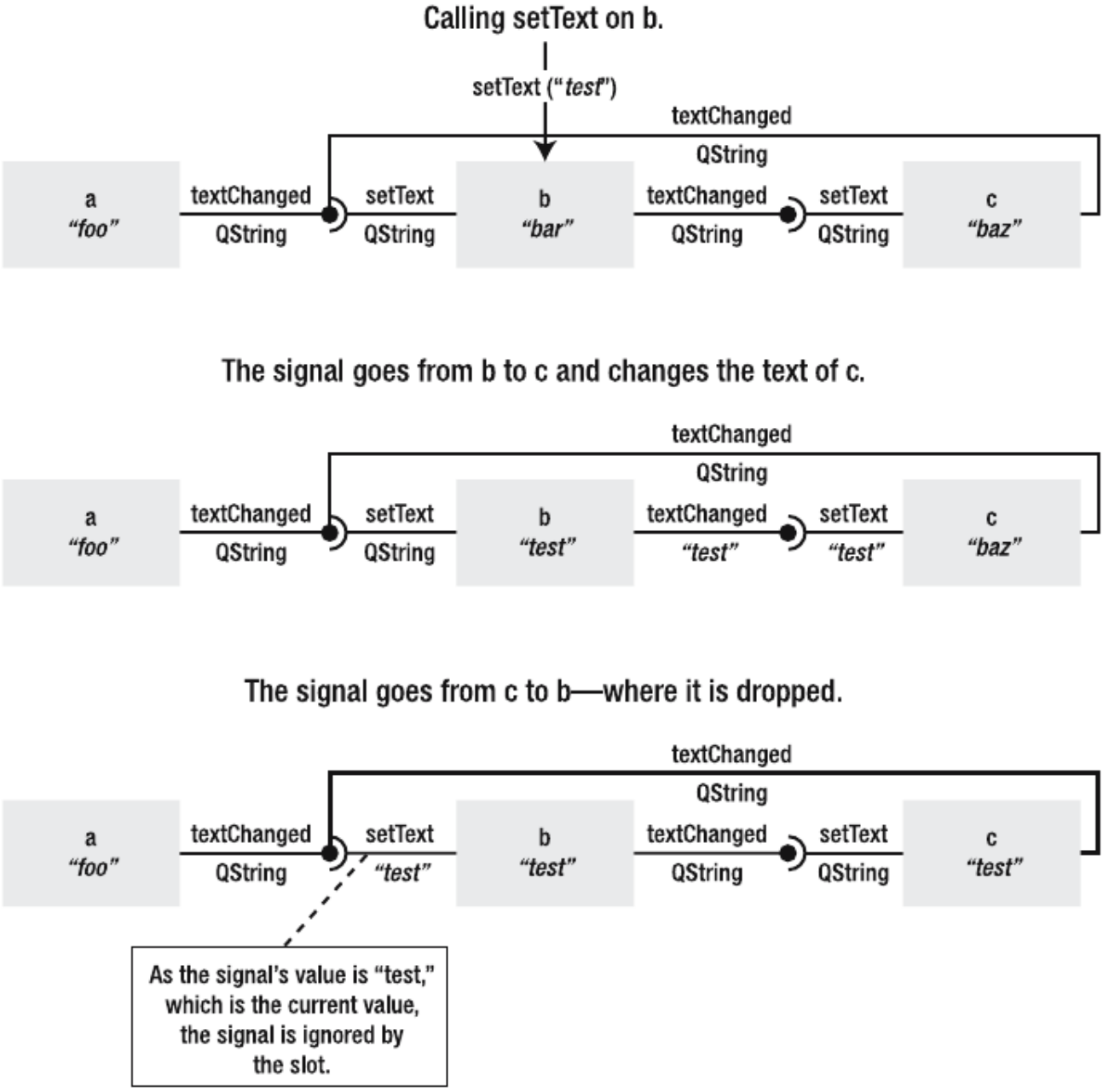


دستور زیر برای نمونه، مقداردهی به یکی از اشیاء را نشان می‌دهد:

```
b->setText( "test" );
```

به نظر شما، اجرای این دستور، باعث تغییر مقدار کدام اشیاء خواهد شد؟

در عکس زیر نحوه‌ی عمل‌کرد سیگنال و اسلاتها بعد از اجرای دستور را می‌بینید:



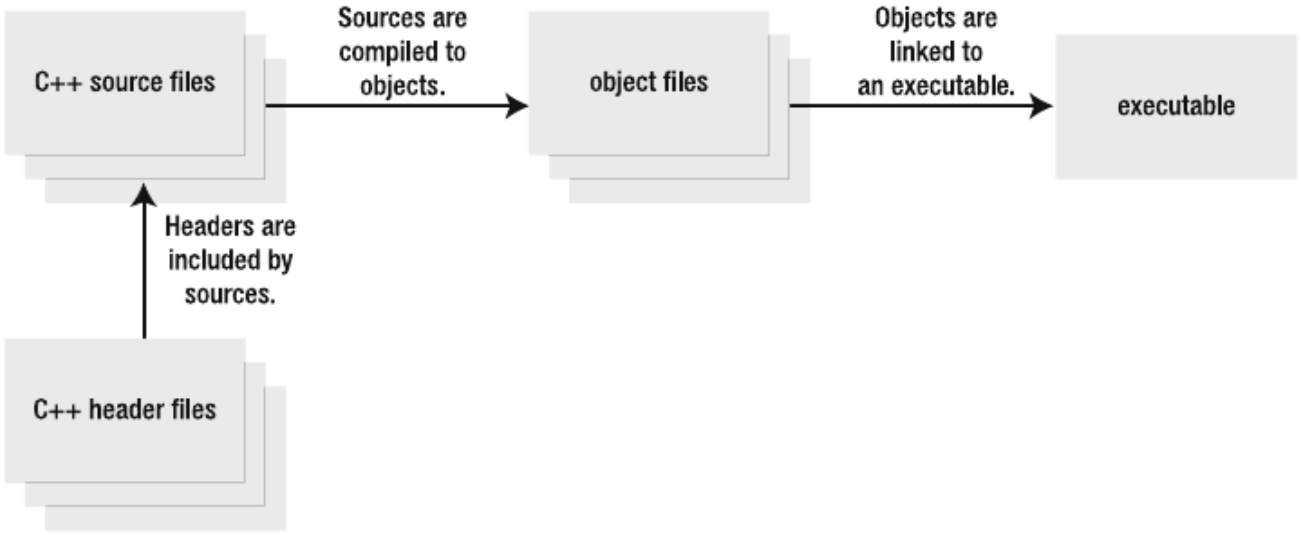
حالا دستور زیر چه تغییری در مقادیر اشیاء خواهد داد؟

```
a->setText( "Qt" );
```

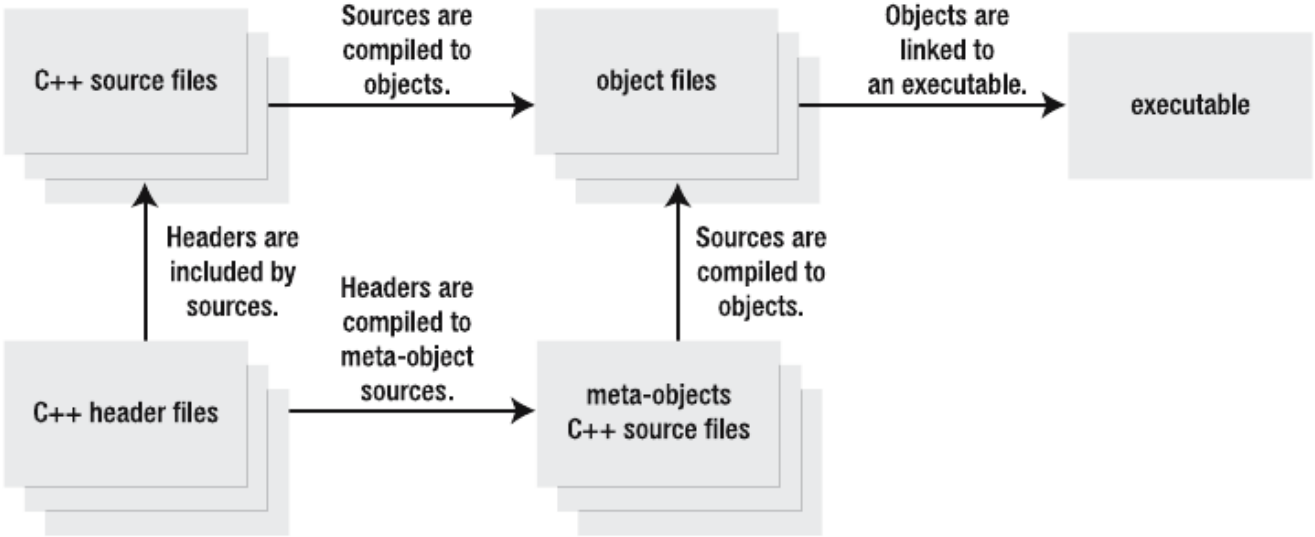
بررسی دوباره‌ی پروسه‌ی کامپایل برنامه

دفعه‌ی پیش که استفاده از qmake را توصیه کردیم، دلیل اینکار عدم وابستگی به سیستم‌عامل بود. دلیل مهم دیگر برای استفاده از آن، این است که qmake پروسه‌ی تولید کدهای تعریفی(متا) را مدیریت می‌کند، و از این بابت خیالمان راحت خواهد بود.

عکس زیر پروسه‌ی کامپایل یک برنامه‌ی C++ را نشان می‌دهد.



زمانی که از qmake استفاده می‌کنیم، همه‌ی header فایلها بوسیله‌ی moc بررسی می‌شوند. moc بدنبال کلاسهایی که ماکروی Q_OBJECT را داشته باشند، می‌گردد و برای آن‌ها فایلهای متا آبجکت را تولید می‌کند، و در نهایت در زمان لینک آن‌ها را به دیگر فایلها لینک می‌کند، در عکس زیر ببینید:



نکته: بیاد داشته باشید که Qt همان C++ است، بعلاوه‌ی تعدادی ماکرو و ابزار تولیدکننده‌ی کد moc .
اگر در زمان کامپایل خطاهایی در زمینه‌ی یافت نشدن signal ها دریافت کردید، به احتمال زیاد فراموش کرده‌اید ماکروی Q_OBJECT را در تعریف کلاس قرار دهید. گاهی نیز خطاهایی بدلیل قراردادن این ماکرو و عدم ارث‌بری کلاس از QObject دریافت خواهید کرد.

یک مثال دیگر

در زمان اتصال دو سیگنال و اسلات هیچ نیازی نیست که سیگنال یا کلاس دارنده‌ی آن یا اسلات در مورد کلاسی که به آن متصل می‌شوند، اطلاعی داشته باشند، صرفاً هماهنگی بین آرگومانهای دو تابع کفایت می‌کند.
حال یک مثال دیگر با واسط دیداری می‌زنیم:
قصد ما بر این است که کاربر متنی را در یک فیلد (اینجا از QLineEdit استفاده می‌کنیم) وارد کند، و همان متن را در یک فیلد دیگر(اینجا از QLabel استفاده می‌کنم.) که قابل ویرایش نیز نیست، ببیند. و کلاس ما بعنوان پلی بین این دو ویدجت عمل می‌کند.
در عکس زیر نحوه‌ی این عمل را ملاحظه می‌کنید:

تابع main برنامه به این شکل خواهد بود:

```
1 #include <QtGui>
2 int main( int argc, char **argv )
3 {
4   QApplication app( argc, argv );
5   QWidget widget;
6   QLineEdit *lineEdit = new QLineEdit;
7   QLabel *label = new QLabel;

8   QVBoxLayout *layout = new QVBoxLayout;
9   layout->addWidget( lineEdit );
10  layout->addWidget( label );
11  widget.setLayout( layout );

12  MyClass *bridge = new MyClass( "", &app );

13  Qobject::connect(lineEdit, SIGNAL(textChanged(constQString&)),
14                bridge, SLOT(setText(const QString&)) );
15  Qobject::connect( bridge, SIGNAL(textChanged(const QString&)),
16                 label, SLOT(setText(const QString&)) );

17  widget.show();
18  return app.exec();
19 }
```

در اینجا برای بهتر شدن ظاهر برنامه از Layout های کیوت استفاده کردیم. (خطوط ۸ تا ۱۱) که در آینده مفصل‌تر درباره‌ی آن‌ها خواهیم گفت.

پیاده‌سازی تابع main را به ۵ بخش تقسیم کرده‌ایم که به مرور بررسی می‌کنیم:

بخش اول که شامل تعاریف اولیه‌ی برنامه، تعریف ویدجت widget برای استفاده بعنوان پنجره‌ی اصلی برنامه و تعریف دو ویدجت مورد نیاز است. همانطور که متوجه شدید در این بخش برای دو ویدجت label و lineEdit ویدجت والد را مشخص نکردیم. در ادامه(بخش دوم) که این ویدجتها را بعنوان ویدجتهای پنجره‌ی اصلی به آن افزودیم، بطور خودکار مقدار parent این ویدجتها تنظیم می‌گردد. (و البته که می‌توانستیم، و بهتر بود که همینجا تنظیم کنیم. و اگر اینجا والد آن‌ها را `app` تعیین می‌کردیم، در زمان افزوده شدن به ویدجت widget والد آن‌ها تغییر می‌کرد.)

از بخش دوم فعلاً بگذریم.

در بخش سوم هم یک نمونه(instance) از کلاس MyClass که می‌خواهیم بعنوان پل از آن استفاده کنیم، تعریف شده است.

بخش چهارم ارتباط بین اشیاء را تعریف می‌کند.

و بخش آخر که پنجره را نمایش داده و وارد حلقه‌ی رخداد(event loop) برنامه می‌شویم.

۳ نکته در مورد برنامه:

نکته ۱: خط اول برنامه، می‌بینیم که فایل QtGui را include کرده‌ایم. Qt برای سادگی کار، هدرفایلهای هر ماژولش را در یک فایل به نام همان ماژول مثلاً QtXml ، QtGui ، QtCore و ... آورده است، و برنامه‌نویس می‌تواند در صورت علاقه از این فایلها استفاده نماید. اما از نظر فنی و بهینه‌سازی اینکار بهینه نیست، و زمان کامپایل و حجم فایل object را افزایش می‌دهد. سعی کنید از آن‌ها استفاده نکنید، و بجای آن همان کلاسها که استفاده شده‌اند را include کنید.

نکته ۲: همانطور که حدس زدید، در نوشتن این برنامه نیازی به کلاس MyClass نبود، و می‌توانستیم بطور مستقیم دو ویدجت را بهم متصل کنیم D **:**

نکته ۳: همانطور که ملاحظه نمودید، کلاس MyClass هیچ اطلاعی در مورد QLabel و QLineEdit ندارد، و اصولاً نیازی ندارد. فقط لازم است کلاسها وارث خصوصیات QObject باشند. باقی اطلاعات در زمان اجرا از meta- object استخراج می‌شود.

برای اجرای برنامه، فایل سورس و هدر را نیز باید به فایل پروژه (pro.) اضافه نمایید. و بعد از آن qmake و make را اجرا کنید.

فصل سوم: Containers and Iterators

کیوت کلاس‌هایی برای جایگزینی بعضی از کلاسهای استاندارد C++ دارد. قبل از این شما با QString آشنا شدید. در این بخش نگاهی به Containerها یا نگهدارنده‌هایی که کیوت برای برنامه‌نویس فراهم کرده است می‌اندازیم.

یکی از مهمترین و مفیدترین ابزارها در برنامه‌نویسی نگهدارنده‌ها هستند، اشیائی که توانایی نگهداری و مدیریت اشیاء دیگر را فراهم می‌کنند.

نگهدارنده‌ها یا Containerهای کیوت، کلاسهای الگویی(template classes) هستند! که می‌توانند هر کلاس تغییرپذیری را نگهدارند. یک بازه از نگهدارنده‌های مختلف مثل لیست(list)، استَک(stack)، صف(queue)، نگاشت(map) و ... در اختیار شما هستند. برای استفاده ساده‌تر و سریعتر از این کلاسها iteratorهایی نیز در اختیار برنامه‌نویس هستند، هم iteratorهای سازگار با روش STL (در صورتی که با STL آشنایی ندارید، زیاد مهم نیست، اشاره به نام آن جهت افراد آشنا می‌باشد.) و هم نسخه‌ی شبیه جاوا.

Iteratorها اشیاء کوچک و سبکی هستند که برای حرکت کردن داخل نگهدارنده‌ها و دسترسی به اشیاء آن‌ها استفاده می‌گردند.

نکته: همه‌ی نگهدارنده‌های کیوت، بطور ضمنی اشتراکی(share) هستند، یعنی تا زمانی که مقداری در یک لیست (بعنوان مثال) تغییر نکند، صرفاً اشاره‌گر به آن بین نمونه‌ها جابجا می‌شود، و کپی نمی‌گردد! پس پاس دادن یک لیست به یک تابع، و یا برگرداندن یک لیست از یک تابع، یک کار سنگین نیست! و پاس دادن const بهمراه آن، آنرا کم هزینه‌تر می‌کند، زیرا تضمین می‌کند که این شیئ هرگز تغییر نخواهد کرد.

بررسی QList

با بررسی کلاس QList آغاز می‌کنیم، همانطور که از اسم کلاس می‌آید، این کلاس برای نگهداری یک لیست از یک شیئ استفاده می‌شود، (با استفاده از این کلاس، شما آرایه‌ها را فراموش خواهید کرد ;))

کد زیر یک لیست از QString را نمایش می‌دهد که با استفاده از اپراتور >> به آن اشیاء جدید اضافه می‌کنیم:

```
QList<QString> list;
list << "foo" << "bar" << "baz";

foreach( QString s, list )
    qDebug() << s;
```

کد نشان می‌دهد توسعه‌دهندگان کیوت چه نظری در مورد یک لیست داشتند، و نحوه‌ی استفاده از آن، ساده و سریع. در C++ دستوری بنام foreach نداریم، اما یک ماکرو در Qt تعریف شده که اینکار را با استفاده از Iteratorها انجام می‌دهد. پس فقط کد را کوتاه و ساده می‌کند برای برنامه‌نویس.

کد زیر هر دو Iterator جاوا و STL را نشان می‌دهد: (کیوت هر دو را پشتیبانی می‌کند.)

```
QList<int> list;
list << 23 << 27 << 52 << 52;

//Iterating through list using Java like iterator:
QListIterator<int> javaIter( list );
while( javaIter.hasNext() )
    qDebug() << javaIter.next();

//Iterating through list using STL like iterator:
QList<int>::const_iterator stlIter;
for( stlIter = list.begin(); stlIter != list.end(); ++stlIter )
    qDebug() << (*stlIter);
```

برای مقایسه‌ی این دو مدل، باید گفت مدل STL بهینه‌تر است. هرچند، سادگی و خوانایی کد شبیه جاوا، می‌تواند دلیل خوبی برای استفاده از آن باشد.

نکته: کاملاً طبیعی و معقول است که از typedef برای جلوگیری از نوشتن چندباره‌ی QList<>::Iterator استفاده کنید. بعنوان مثال یک لیست از MyClass را MyClassList نامگذاری می‌کنیم، اینگونه:

```
typedef QList<MyClass>::Iterator MyClassListIterator;
```

این عمل باعث بالا رفتن خوانایی کد شبیه STL می‌گردد.

در کدی که بررسی کردیم، مقادیر لیست قابل تغییر نبودند، این باعث بهینه‌تر شدن برنامه، در زمانهایی که واقعاً نیازی به تغییر مقادیر لیست نداریم می‌گردد. اما گاهی نیاز به تغییر در لیست داریم، در این مواقع از کد زیر استفاده می‌کنیم:

```
QList<int> list;
list << 27 << 33 << 61 << 62;

QMutableListIterator<int> javaIter( list );
while( javaIter.hasNext() )
{
    int value = javaIter.next() + 1;
    javaIter.setValue( value );
    qDebug() << value;
}

Qlist<int>::Iterator stlIter;
for( stlIter = list.begin(); stlIter != list.end(); ++stlIter )
{
    (*stlIter) = (*stlIter)*2;
    qDebug() << (*stlIter);
}
```

در مورد STL-Iterator اینبار از نام Iterator استفاده کردیم، بجای iterator، این دو مترادف هم هستند، و هردو قابل قبول هستند.

توجه: اینجا مقدار اشیاء داخل لیست را تغییر دادیم، دقت داشته باشید که زمانی که به لیست شیئ‌ای اضافه یا کم می‌کنیم، ممکن است Iterator ما بی‌اعتبار(invalid) گردد. به این معنی که قابل استفاده نباشد.

علاوه بر جلو رفتن با اپراتور ++ شما می‌توانید از اپراتور -- هم برای عقب رفتن در لیست STL استفاده نمایید. همینطور در مورد جاوا توابع previous و findNext و findPrevious هم در دسترس هستند. سعی کنید زمانهای جلو، عقب رفتن در لیست از hasNext و hasPrevious برای اطمینان خاطر از اینکه وضعیت نامشخصی رخ ندهد، استفاده کنید.

نکته: زمانی که می‌خواهید از Iteratorها استفاده کنید، سعی کنید تا جای ممکن از constant iteratorها استفاده کنید، چون هم سرعت بیشتری دارند، هم مطمئن هستید که تغییری در مقادیر ایجاد نخواهد شد.

علاوه بر روشهای بالا، شما می‌توانید از اپراتوی [] یا تابع at برای دسترسی به اشیاء یک لیست استفاده نمایید. کد زیر را ببینید:

```
int sum = list[5] + list.at(7);
```

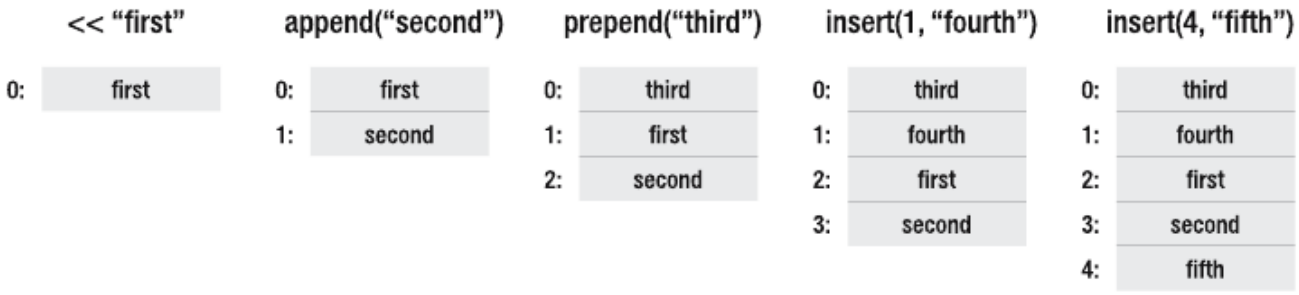
کاملا شبیه یک آرایه، نه؟

پر کردن لیست

تا به اینجا فقط با اپراتور >> برای افزودن مقدار به لیست آشنا شدید، که به معنی افزودن یک مقدار به ته لیست می‌باشد، اما گاهی اوقات ما نیاز داریم یک مقدار جدید را به سر لیست اضافه کنیم، و یا در یک جای خاص، برای این موارد توابع prepend و insert در دسترس هستند:

```
QList<QString> list;
list << "first";
list.append( "second" );
list.prepend( "third" );
list.insert( 1, "fourth" );
list.insert( 4, "fifth" );
```

برای درک بهتر کد، عکس زیر را ببینید:



لیستهای بیشتر

QList تنها کلاس لیست نیست، کلاسهای دیگری برای سناریوهای مختلف موجودند. زمانی که می‌خواهیم کلاس لیستی برای استفاده انتخاب کنیم، در اکثر موارد بهترین انتخاب QList است، تنها ایراد استفاده از QList این است که در مواقعی که می‌خواهیم مقادیر جدید را به وسط لیست اضافه کنیم، کند عمل می‌کند.

دو کلاس ویژه‌ی لیست دیگر داریم، مورد اول QVector است. که تضمین می‌دهد که مقادیر به ترتیب در حافظه نگهداری گردند. پس زمانی که شما یک مقدار به سر لیست، و یا وسط آن اضافه می‌کنید، همه‌ی مقادیر جابجا خواهند شد. اما فایده‌ی QVector زمانی است که می‌خواهیم به ترتیب به مقادیر لیست دسترسی داشته باشیم.

گزینه‌ی دیگر کلاس QLinkedList است، که یک لینک لیست ارائه می‌دهد. اما هیچ تضمینی در مورد ترتیب ذخیره‌ی مقادیر و ترتیب دسترسی به آن‌ها وجود ندارد. زمان افزودن مقدار جدید به لیست، بدون توجه به اینکه کجای لیست باید اضافه گردد یکسان است. که در مورد مشکلی که با QList داشتیم(کندی افزودن مقدار جدید به وسط لیست)، گزینه‌ی خوبی است. خوبی مهم دیگری که لینک لیستها دارند، این است که تا زمانی که مقداری در لیست باشد، Iteratorها مقدار نامشخصی نخواهند داشت. هرچقدر که مقدار اضافه یا کم کنیم.

در جدول زیر سه کلاس را باهم مقایسه کرده‌ایم:

کلاس	افزودن در ابتدای لیست	افزودن در وسط لیست	افزودن در انتهای لیست	دسترسی با اندیس	دسترسی با Iterator
QList	سریع	خیلی کند در لیستهای بزرگ	سریع	سریع	سریع
QVector	کند	کند	سریع	سریع	سریع
QLinkedList	متوسط	متوسط	متوسط	امکان ندارد	سریع

لیستهای مخصوص

تا به اینجا لیستهای عمومی را دیدیم، اما کیوت لیستهایی مخصوص یک داده‌ی خاص نیز دارد، بعنوان مثال لیستی از رشته‌ها را بررسی می‌کنیم: QStringList

این کلاس از QList<QString> ارثبری نموده‌است، پس کاملاً مثل آن قابل استفاده است، اما توابع و امکاناتی بیشتر نیز دارد، که کار با رشته‌ها را ساده‌تر می‌کند:

در ابتدا نیاز به یک لیست داریم:

```
QStringList list;
list << "foo" << "bar" << "baz";
```

حال یک لیست داریم که داخل آن رشته‌های bar ، foo و baz نگهداری می‌گردند.

شما می‌توانید مقادیر آنرا با یک رشته‌ی دلخواه خود بهم بچسبانید و یک رشته‌ی دیگر تولید کنید:

```
QString all = list.join(",");
```

حال رشته‌ی all مقدار "foo,bar,baz" را خواهد داشت. کار دیگری که بعنوان مثال می‌توانید انجام دهید، جایگزین کردن یک رشته با رشته‌ای در همه‌ی مقادیر لیست، بعنوان مثال می‌خواهیم کاراکتر 'a' را با 'oo' جایگزین نماییم.

```
list.replaceInStrings( "a", "oo" );
```

علاوه بر join کلاس QString یک تابع بنام split دارد، که کار آن برعکس است، یعنی براساس رشته‌ای که به تابع split می‌دهیم، رشته‌ی ما را تکه‌تکه می‌کند، و نتیجه را در یک QStringList برمی‌گرداند:

```
list << all.split(",");
```

به نظرتان list الان چه مقادیری دارد؟

درست حدس زدید؟

"foo", "boor", "booz", "foo", "bar", "baz"

پشته‌ها و صف‌ها

حال می‌خواهیم با کلاسهای ویژه‌ای که برای افزودن مقادیر جدید در محل خاصی از لیست، و برداشتن مقدار از محل خاص دیگری بهینه‌شده‌اند آشنا شویم.

کلاسهای QQueue و QStack .(با مفاهیم پشته و صف آشنایی دارید؟)

پشته‌ها لیست‌هایی هستند که بعنوان LIFO شناخته می‌شوند یا Last In, First Out که آخرین مقداری که به آن‌ها افزوده‌اید اولین مقداری است که می‌توانید از آن بردارید.

صف‌ها نیز، لیستهای FIFO شناخته می‌شوند (First In First Out) که شبیه یک صف عمل می‌کنند، هر مقداری که زودتر وارد صف شده باشد، زودتر از آن خارج می‌شود.

یک پشته چهار تابع مهم و کاربردی دارد:

push برای افزودن مقدار جدید به پشته استفاده می‌شود.

top برای گرفتن و داشتن مقدار بالای پشته استفاده می‌شود.

pop برای حذف کردن مقدار بالای پشته و برگرداندن و استفاده از آن.

isEmpty نشان می‌دهد که آیا پشته خالی است یا نه!؟

کد زیر را ببینید:

```
QStack<QString> stack;

stack.push( "foo" );
stack.push( "bar" );
stack.push( "baz" );

QString result;
while( !stack.isEmpty() )
    result += stack.pop();
```

در مورد صف، این توابع به شرح زیر هستند:

enqueue برای افزودن مقدار جدید به ته صف.

dequeue برای برداشتن و حذف مقدار سر صف.

head برای دسترسی به مقدار سر صف.

isEmpty بررسی اینکه آیا صف خالی است یا نه!؟

کد زیر استفاده از این توابع را نشان می‌دهد:

```
QQueue<QString> queue;
queue.enqueue( "foo" );
queue.enqueue( "bar" );
queue.enqueue( "baz" );
QString result;
while( !queue.isEmpty() )
    result += queue.dequeue();
```

نگاشت (Map)

لیستها برای نگهداری اشیاء خوب هستند، اما گاهی اوقات نیاز به نگه‌داری اشیاء به طرق دیگری داریم. چیزی شبیه به یک دیکشنری، یک نگاشت بین یک کلید و یک مقدار! اینجاست که کلاسهای QMap و QHash وارد تصویر می‌شوند. (اگر به پایتون آشنا باشید، چیزی شبیه به dict هستند.)

زمانی که یک QMap می‌سازید، کلاسهای الگویی که به آن می‌دهید به ترتیب نوع کلید و نوع مقدار هستند:

```
QMap<QString,int> ourMap;
map["foo"] = 42;
map["bar"] = 13;
map["baz"] = 9;
```

برای افزودن یک مقدار به یک نگاشت(map) تنها چیزی که لازم است همین روش بالا است، اینگونه اگر کلید مورد نظر در نگاشت وجود نداشته باشد، یک کلید/مقدار اضافه می‌گردد، و اگر وجود داشته باشد، مقدار آن تغییر می‌کند و مقدار جدید را می‌گیرد.

شما می‌توانید با استفاده از تابع contains بررسی کنید که آیا این کلید در نگاشت موجود است یا نه!؟

نمایش دادن همه‌ی جفتی‌های کلید/مقدار:

```
foreach( QString key, map.keys() )
    qDebug() << key << " = " << map[key];
```

اگر یادتان باشد، گفتیم ماکروی foreach از Iteratorها برای حرکت روی یک نگهدارنده استفاده می‌کند. پس ما هم می‌توانیم بطور مستقیم، از یک Iterator برای حرکت و دستیابی به کلید/مقدارهای یک نگاشت استفاده کنیم:

```
QMap<QString, int>::ConstIterator ii;
for( ii = map.constBegin(); ii != map.constEnd(); ++ii )
    qDebug() << ii.key() << " = " << ii.value();
```

عملگر [] برای دستیابی به مقادیر موجود در نگاشت خوب است، اما همانطور که گفتیم، اگر کلید مورد نظر موجود نباشد، آن کلید ساخته می‌شود. و بعنوان مثال در کد زیر، یک کلید/مقدار جدید با کلید مورد نظر ما ساخته می‌شود و مقدار آن مقدار پیشفرض خواهد بود:

```
sum = map["foo"] + map["ingenting"];
```

اما در اینجا منظور ما این نیست!

پس بهتر است استفاده از تابع value بجای اپراتور [] را تمرین کنید. در این صورت یک مقدار پیشفرض ساخته و برگردانده می‌شود، اما به Map ما اضافه نمی‌شود:

```
sum = map["foo"] + map.value("ingenting");
```

زمانی که یک شیئ نگاشت می‌سازیم، نوع داده‌ای که بعنوان کلید به آن می‌دهیم باید از اپراتورهای == و < پشتیبانی کند. چون نگاشت باید بتواند کلیدها را با هم مقایسه و مرتب کند. QMap جستجوی خوبی دارد، زیرا همیشه کلیدها را مرتب نگه‌می‌دارد، نه به ترتیبی که آن‌ها را وارد نگاشت کرده‌ایم.

شما می‌توانید حتی نتیجه‌ی بهینه‌تری داشته باشید با استفاده از کلاس QHash بجای QMap زیرا کلاس QHash کلیدها را مرتب نمی‌کند.

(بحث QHash که در ادامه می‌آید ممکن است کمی سنگین و گیج‌کننده باشد، می‌توانید فعلا از آن عبور کنید.)

QHash می‌تواند به همان طریق QMap استفاده شود، با این تفاوت که نوع داده‌ای که به آن بعنوان کلید می‌دهید باید اپراتور == داشته باشد و یک تابع عمومی بنام qHash ، که تابع qHash باید یک مقدار unsigned int برگرداند که به آن hash key یا کلید hashing می‌گوییم، که برای هر کلید منحصربه فرد است، اینگونه سرعت جستجو بیشتر می‌شود. کیوت این تابع را برای اکثر نوع داده‌ای ها آماده کرده است! فقط در صورتی که می‌خواهید کلاسی که خود نوشته‌اید را در یک QHash نگهدارید، لازم است این تابع را آماده کنید.

کارایی hash list ها بستگی به تعداد برخورد(تصادم)هایی که می‌تواند انتظار داشته باشد دارد. که تعداد کلیدهایی است که کلید hashing یکسان خواهند داشت. با استفاده از اطلاعاتی که در مورد کلید دارید، می‌توانید کارایی آنرا افزایش دهید. بعنوان مثال در یک برنامه‌ی دفترچه تلفن، افراد ممکن است اسامی یکسانی داشته باشند. اما ممکن نیست نام یکسان و شماره تلفن یکسان داشته باشند، از همین مسأله برای حل مشکل تصادم استفاده می‌کنیم.

کد زیر، کلاس فرد را که نام و شماره تلفن را نگه می‌دارد را نشان می‌دهد:

```
class Person
{
public:
Person( const QString& name, const QString& number );
const QString& name() const;
const QString& number() const;
private:
QString m_name, m_number;
};
```

برای این کلاس، شما باید یک اپراتور == و یک تابع qHash آماده نمایید.

برای تابع qHash، ما مقدار hash اسم و شماره را از qHash خود کیوت می‌گیریم، و نتایج را با استفاده از اپراتور منطقی XOR (^)جمع می‌بندیم:

```
bool operator==( const Person &a, const Person &b )
{
    return (a.name() == b.name()) && (a.number() == b.number());
}

uint qHash( const Person &key )
{
    return qHash( key.name() ) ^ qHash( key.number() );
}
```

برای آزمایش تابع qHash یک لیست می‌سازیم و مقادیری به آن می‌افزاییم، سپس وجود و عدم وجود مقادیری را در آن می‌آزماییم:

```
QHash<Person, int> hash;

hash[ Person( "Anders", "8447070" ) ] = 10;
hash[ Person( "Micke", "7728433" ) ] = 20;

qDebug()<< hash.value( Person("Anders", "8447070" ) ); // 10
qDebug()<< hash.value( Person("Anders", "8447071" ) ); // 0
qDebug()<< hash.value( Person("Micke", "7728433" ) ); // 20
qDebug()<< hash.value( Person("Michael", "7728433" ) ); // 0
```

فصل چهارم: استفاده از QMake

qmake ابزاری برای ساده و یکسان سازی پروسه کامپایل برنامه ها در محیطهای گوناگون می‌باشد. در واقع برنامه qmake فایلهای Makefile مورد نیاز برای کامپایل برنامه را تولید می‌کند. اما به سادگی نوشتن تنها چند خط برای کاربر. همچنین امکانات ویژه ای برای برنامه‌های نوشته شده با کیوت را دارا می‌باشد مانند افزودن دستورات مورد نیاز برای استفاده از moc و uic.

این راهنما نکات پایه‌ی استفاده از ابزار qmake را در بر می‌گیرد، برای اطلاعات بیشتر به مستندات Qt مراجعه نمایید.

شروع ساده:

در نظر بگیرید، کد اولیه برنامه مورد نظر را نوشته و می‌خواهیم آنرا تست کنیم. و حالا ما سه فایل شامل کدهای برنامه داریم:

- hello.cpp
- hello.h
- main.cpp

نکته‌ی دیگری که در مورد برنامه می‌دانیم این است که برنامه با استفاده از Qt نوشته شده است.

در نظر بگیریم که هر سه فایل کد ما در پوشه‌ی project قرار دارند:

وارد پوشه project شده، بوسیله‌ی ویرایشگر متن مورد علاقه تان، فایلی بنام hello.pro ایجاد نمایید. که در واقع همان فایلی است که به عنوان ورودی به qmake خواهیم داد.

حال باید به qmake بگوییم فایلهای سورس برنامه کدامهاست؟

برای اینکار ما از متغیر از پیش تعریف شده SOURCES استفاده می‌کنیم، و مقادیر مورد نظر را به آن می‌دهیم:

برای اینکار خطی را با SOURCES آغاز نموده اسم فایل hello.cpp را به آن مقدار دهی می‌نماییم:

```
SOURCES += hello.cpp
```

و اینکار را در مورد همه فایلهای سورس برنامه تکرار می‌کنیم، پس در پایان فایل hello.pro به این شکل خواهد بود:

```
SOURCES += hello.cpp
SOURCES += main.cpp
```

اگر علاقه مند به لیست کردن همه فایلها بصورت یکجا هستید، می‌توانید از روش زیر نیز استفاده نمایید:

```
SOURCES = hello.cpp \
    main.cpp
```

حال که سورس فایلها مشخص شدند، هدر(header) فایلها را نیز با استفاده از متغیر HEADERS مشخص می‌کنیم، به اینصورت که خط زیر را به فایل hello.pro می‌افزاییم:

```
HEADERS += hello.h
```

نام فایل اجرایی خروجی، بصورت خودکار از نام فایل .pro تعیین خواهد شد، یعنی مثلا اینجا، خروجی برنامه در سیستم عامل ویندوز hello.exe و در گنو/لینوکس hello خواهد بود. اگر می‌خواهید این رفتار را تغییر دهید، به شیوه زیر عمل نمایید:

```
TARGET = helloworld
```

حال فایل اجرایی در پایان کامپایل helloworld.exe خواهد شد.

آخرین بخش، تعیین مقدار متغیر CONFIG خواهد بود. از جایی که این برنامه با استفاده از توابع و کلاسهای کیوت نوشته شده است، باید qt را در تنظیمات تعیین نماییم، تا qmake کتابخانه های مربوطه را به برنامه لینک نموده و دستورات مربوط به moc و uic را در فایل Makefile خروجی بیافزاید.

در نهایت فایل hello.pro به این شکل خواهد بود:

```
CONFIG += qt
HEADERS += hello.h
SOURCES += hello.cpp
SOURCES += main.cpp
```

شما حالا می‌توانید با استفاده از برنامه qmake یک Makefile برای کامپایل برنامه خود بسازید:

```
qmake -o Makefile hello.pro
```

سپس برای کامپایل برنامه make را اجرا کنید.

برای کاربران ویژوال استدیو، qmake می‌تواند فایل .dsp و یا .vcproj تولید کند، به این صورت:

```
qmake -tp vc -o hello.dsp hello.pro
```

چطور خروجی برنامه را قابل اشکال زدایی(debug) کنیم؟

پر واضح است که فایل اجرایی برنامه بطور پیش فرض قابلیت اشکال زدایی را ندارد، چون برای این منظور، اطلاعات بیشتری باید در زمان کامپایل به فایل اجرایی اضافه شود.

قابل دیباگ کردن فایل اجرایی خروجی، به سادگی افزودن debug به مقدار متغیر CONFIG می‌باشد.

بطور مثال:

```
CONFIG += qt debug
HEADERS += hello.h
SOURCES += hello.cpp
SOURCES += main.cpp
```

افزودن سورس فایلهایی مخصوص یک محیط خاص

در حین برنامه نویسی، ممکن است کدها یا سورسهایی داشته باشید که بخواهید فقط در یک محیط خاص مثلا ویندوز استفاده شوند.
محتوای فایل proی زیر را ببینید:

```
CONFIG += qt debug
HEADERS += hello.h
SOURCES += hello.cpp
SOURCES += main.cpp
win32 {
  SOURCES += hellowin.cpp
}
unix {
  SOURCES += hellounix.cpp
}
```

در صورتی که این فایل در محیط ویندوز استفاده شود، فایل hellowin.cpp به سورسهای برنامه اضافه خواهد شد و کامپایل خواهد شد، و در صورتی که در محیط یونیکس / لینوکس استفاده شود، فایل hellounix.cpp کامپایل خواهد شد.

متوقف کردن qmake در صورت عدم وجود یک فایل

ممکن است بخواهید، در صورت عدم وجود یک فایل خاص، تولید Makefile را متوقف نمایید. در اینصورت از روش زیر استفاده می‌کنیم:

```
!exists( main.cpp ) {
  error( "No main.cpp file found" )
}
```

پروژه های عمومی *QMake*

در این قسمت، نحوه‌ی تنظیم فایل .pro برای ساختن ۳ نوع پروژه مختلف Qt بوسیله QMake را توضیح خواهیم داد:

1. **ساختن یک برنامه کاربردی**
2. **ساختن یک کتابخانه**
3. **ساختن یک پلاگین**

ساختن یک برنامه کاربردی

قالب app

با تعیین قالب برنامه بعنوان app به qmake اطلاع می‌دهیم که باید Makefile ای که می‌سازد، یک برنامه کاربردی (Application) تولید کند.

اینکار بوسیله افزودن خط زیر به فایل .pro انجام میگیرد:

```
TEMPLATE = app
```

با این قالب نوع برنامه به یکی از دو صورت زیر می‌تواند تعیین شود: (با افزودن آن به مقدار متغیر CONFIG)

مقدار	توضیح
windows	برنامه مورد نظر واسط گرافیکی دارد.
console	برنامه مورد نظر واسط گرافیکی ندارد، و اصطلاحا خط فرمانی می‌باشد

زمانی که قالب برنامه را app تعیین کردیم، متغییرهای زیر قابل تشخیص و استفاده در فایل .pro خواهند بود:

متغیر	توضیح
HEADERS	لیستی از همه هدر فایلهای استفاده شده در برنامه.
SOURCES	لیستی از همه سورس فایلهای استفاده شده در برنامه
FORMS	لیست همه فایلهای .ui تولید شده توسط برنامه طراح و استفاده شده در برنامه.
TARGET	اسم فایل اجرایی نهایی. بصورت پیش فرض اسم فایل .pro استفاده می‌شود.
DESTDIR	پوشه ای که فایل اجرایی نهایی در آن قرار خواهد گرفت.
DEFINES	لیستی از متغییرهای pre-processor ای که در صورت نیاز می‌توانیم برای برنامه تعریف کنیم.
INCLUDEPATH	لیستی از آدرسهای include اضافه برای برنامه، محلهایی که برنامه می‌تواند هدر فایلهای استفاده شده را بیابد.
DEPENDPATH	آدرسی که برنامه به آن وابسته است.
DEF_FILE	مخصوص ویندوز: آدرس یک فایل .def برای استفاده در زمان لینک.
RC_FILE	مخصوص ویندوز: یک فایل منبع (resource) برای برنامه.
RES_FILE	مخصوص ویندوز: آدرس یک فایل منبع جهت لینک شدن به برنامه، هنگام کامپایل

فقط در صورتی نیاز به تنظیم این متغیرها خواهید داشت، که مقداری برای آن داشته باشید، بطور مثال اگر هیچ آدرسی شامل هدرفایلهای اضافه برای برنامه نداشته باشید، نیازی به تنظیم آن نخواهد بود. Qmake آدرسهای پیشفرض سیستم را خودکار به این متغیر اضافه می‌کند.

بعنوان مثال، فایل پروژه‌ی یک برنامه اینگونه می‌تواند باشد:

```
TEMPLATE = app
DESTDIR  = c:/helloapp
HEADERS += hello.h
SOURCES += hello.cpp
SOURCES += main.cpp
DEFINES += QT_DLL
CONFIG  += qt warn_on release
```

برای متغییرهایی که تنها یک مقدار می‌گیرند مثل TEMPLATE ما از مساوی (=) استفاده می‌کنیم، و برای متغیرهای چند مقداره مثل SOURCES از بعلاوه‌مساوی (=+) تا مقدار جدید را به مقادیر موجود بیافزاید. در صورت استفاده از مساوی (=) مقدار کنونی حذف خواهد شد.
بعنوان مثال اگر اینجا بنویسیم DEFINES = QT_DLL همه‌ی تعاریفی که قبلا qmake بطور خودکار تعریف کرده حذف و بلا استفاده خواهند شد.

ساختن یک کتابخانه

قالب lib

با تعیین متغییر TEMPLATE به lib ، qmake متوجه می‌شود که خروجی کامپایل یک برنامه کاربردی نیست، بلکه یک کتابخانه خواهد بود.
در قالب lib علاوه بر متغیرهای ذکر شده ذیل قالب app یک متغیر دیگر نیز داریم:

- VERSION که با تعیین آن، شماره ورژن کتابخانه را تعیین می‌کنیم.

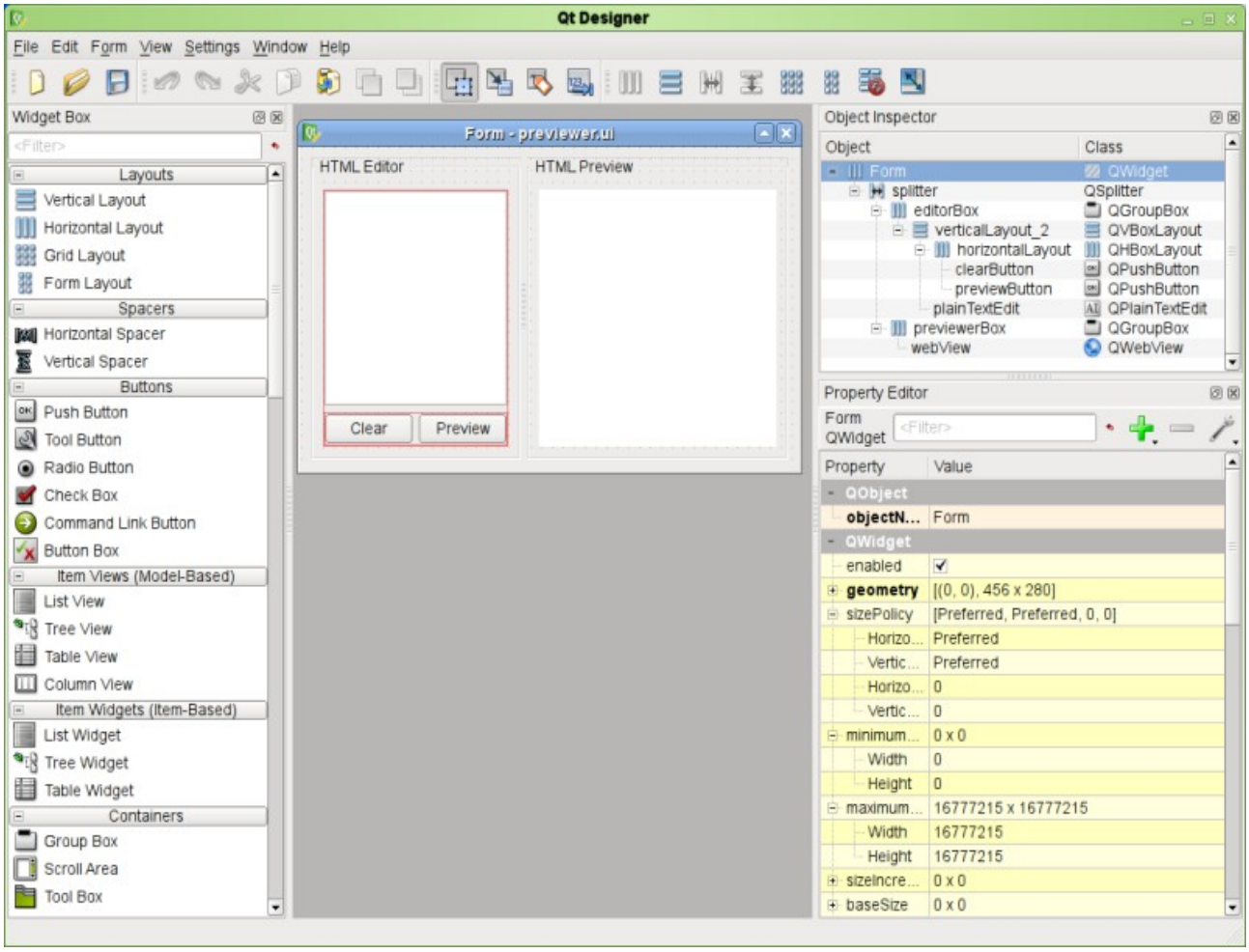
همچنین، متغیر CONFIG را نیز می‌توان به یکی از مقادیر زیر تنظیم کرد:

مقدار	توضیح
dll	یک کتابخانه اشتراکی (Shared Lib)
staticlib	یک کتابخانه ایستا (Static Lib)
plugin	این کتابخانه یک پلاگین است. این مقدار همچنین گزینه dll را نیز فعال می‌کند.

ساختن یک پلاگین

پلاگینها با استفاده از قالب lib ساخته می‌شوند. همانطور که در قسمت قبل توضیح داده شد. مقدار plugin را به متغیر CONFIG می‌افزاییم. و قالب را lib تعریف می‌کنیم.

فصل پنجم: آشنایی با برنامه طراح Qt



اجرای طراح:

در سیستم عامل ویندوز برنامه‌ی طراح را از منوی start و یا فایل اجرایی designer در پوشه‌ی bin محلی که sdk نصب شده است اجرا نمایید.

در سیستم عامل گنو/لینوکس هم در منوهای kde و یا برنامه های Gnome قابل دستیابی است، و البته راه میانبر اجرای دستور designer در خط فرمان است.

واسط برنامه‌ی طراح:

همانگونه که در عکس بالا ملاحظه می‌کنید، ظاهر برنامه شامل چند بخش اصلی می‌باشد:

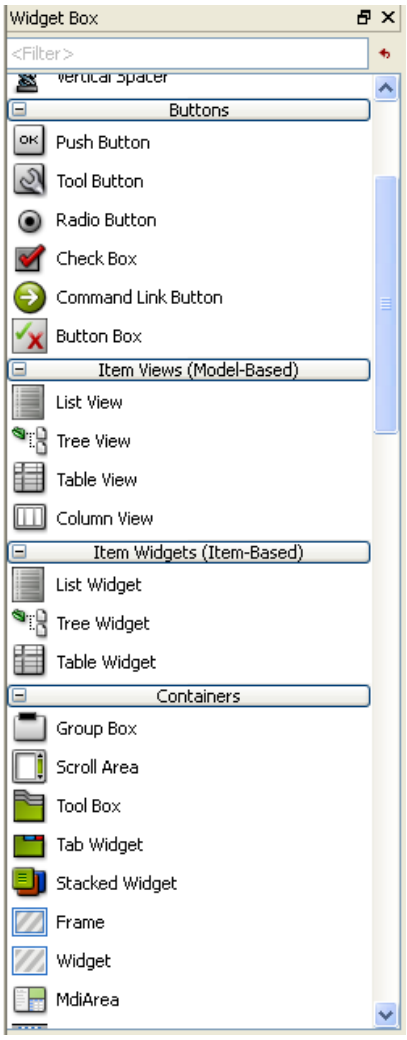
۱. منوها و نوار ابزار برنامه در بالای پنجره.
۲. بخش میانی با پس زمینه‌ی خاکستری که پنجره های در حال ویرایش در آن قرار می‌گیرند.
۳. طرفین پنجره که شامل ابزارهای جانبی ویرایش واسط گرافیکی می‌باشند. که قابل جابجایی در طرفین می‌باشند.

ابزارهای جانبی:

جعبه ابزار:

همانطور که در عکس روبرو ملاحظه می‌کنید، این جعبه شامل اشیا و ابزاری از پیش آماده همانند دکمه های مختلف و... برای استفاده در برنامه‌ی مورد نظر می‌باشد.

نحوه‌ی استفاده از این جعبه، به سادگی گرفتن، کشیدن و انداختن یک ویدجت بر روی فرم مورد نظر می‌باشد!

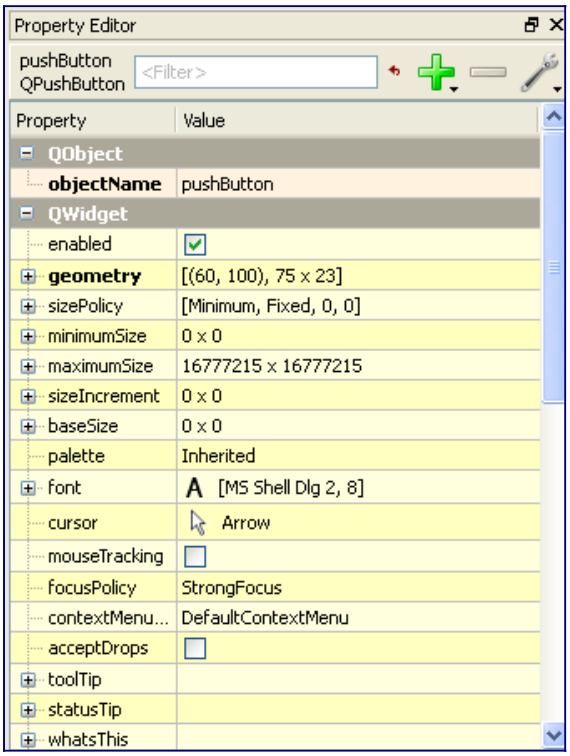


ویرایشگر خصوصیات:

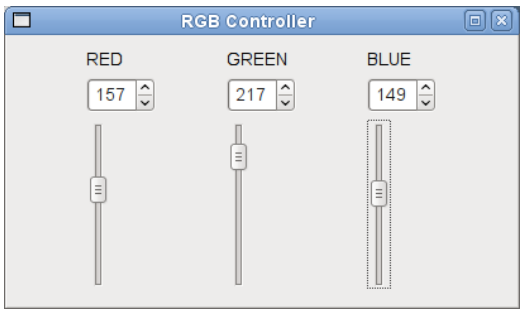
با انتخاب هر یک از ویدجتهای داخل فرم، اطلاعات و خصوصیات آن در این جعبه نمایان می‌شود، که به ما امکان ایجاد تغییرات دلخواه در آنها را می‌دهد.

ویدجت انتخاب شده در تصویر مقابل یک QPushButton است، که بطور مثال با تغییر دادن خصوصیت font، می‌توان فونت نوشته‌ی دکمه را تغییرداد، یا با تغییر مقدار خصوصیت toolTip می‌توان تعیین کرد که چه توضیحاتی را برنامه به هنگامی که نشانگر موس کاربر بر روی این ویدجت قرارگرفت به او نشان دهد.

و الی آخر...



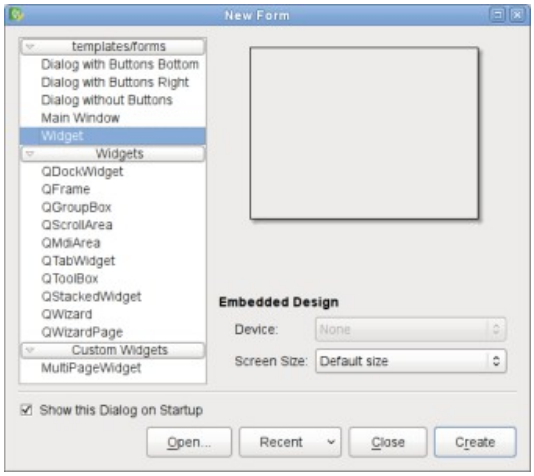
برای اینکه درک کاملتری از نحوه‌ی استفاده از این ابزارها و دیگر ابزارهای موجود در برنامه‌ی طراح Qt داشته باشید، یک فرم ساده را طراحی می‌نماییم:



استفاده از طراح کیوت، شامل حداقل چهار مرحله ساده می‌باشد:

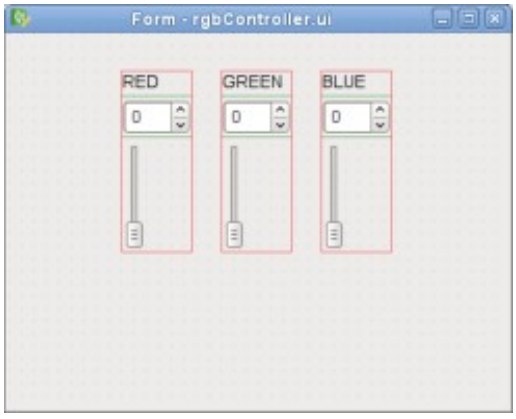
- ❖ انتخاب فرم و اشیاء داخل آن
- ❖ طرح‌بندی و قالب بندی اشیاء داخل فرم
- ❖ اتصال سیگنالهای مورد نظر به اسلاتها (در صورت لزوم)
- ❖ دیدن پیش نمایش فرم و ایجاد تغییرات لازم دیگر

در نظر بگیرید قصد طراحی فرم بالا را داریم، یک فرم ساده که دارای کنترلهایی برای تعیین مقدار رنگهای اصلی در یک عکس می‌باشد، پنجره ای که در برنامه های ویرایشگر عکس یافت می‌شود.



انتخاب فرم:

با انتخاب فرم مورد نظر از پنجره‌ی New Form شروع می‌نماییم. (این پنجره از طریق منوی File->New قابل دستیابی است.)



گذاردن ویدجتها روی فرم:

همانطور که پیشتر اشاره شد، اینکار به سادگی انتخاب، کشیدن و انداختن ویدجت مورد نظر از جعبه ابزار می‌باشد! حالا سه Label(برچسب) و سه Spin Box و سه عدد Slider روی فرم بیاندازید.

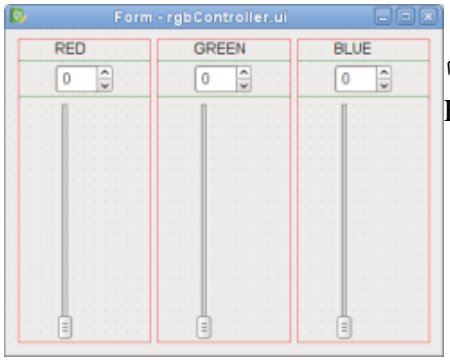
تغییر متن برچسب به سادگی دوبار کلیک کردن و وارد کردن متن دلخواه می‌باشد.

مسلما زمانی که ویدجتها را بروی فرم انداختید، خودبخود به این شکل قرار نمی گیرند! چطور آنها را مرتب کنیم!؟

منوی Form ابزاری برای اینکار دارد. (منوهای مورد نظر، روی نوار ابزار بالای پنجره اصلی برنامه نیز قابل دسترس می‌باشند:

برای اینکار: برچسب RED را انتخاب کرده، سپس با نگهداشتن دکمه‌ی Ctrl روی یک SpinBox و یک Slider کلیک کنید تا هرسه بطور همزمان انتخاب گردند. حالا در منوی Form گزینه‌ی Lay Out Vertically را انتخاب نمایید. ساده بود نه!؟

حالا، اینکار را در مورد دو رنگ دیگر هم تکرار نمایید!



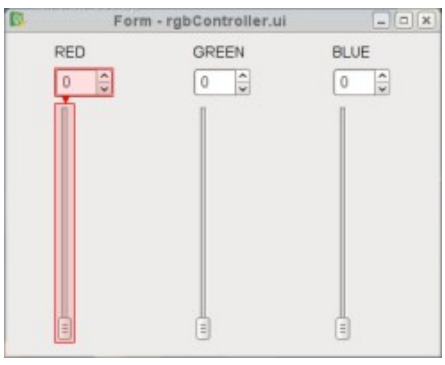
و دست آخر برای کنار هم قرار دادن هرسه قسمت(رنگ) در فرم بصورت افقی، می‌توانید روی فرم کلیک کرده، و از منوی Form گزینه‌ی Lay Out Horizontally را انتخاب نمایید.

در نهایت فرم ما به شکل روبرو در خواهد آمد!

نکته: کل اینکار را بصورت یکجا با استفاده از گزینه Lay Out in Grid هم می‌توانید انجام دهید!

البته که شما می‌توانستید بدون استفاده از لایه ها بطور دستی ظاهر فرم را به این شکل در بیاورید، اما! در آن صورت با تغییر دادن سایز فرم، ویدجتهای شما در همان وضعیت می‌ماندند. مهمترین فایده‌ی لایه ها در کیوت، این است که با تغییر دادن سایز پنجره، ویدجتها و لایه ها هم تغییر اندازه می‌دهند و با اندازه‌ی جدید هماهنگ می‌گردند، بعلاوه اینکه وقتی در زمان اجرا به پنجره ویدجت اضافه می‌کنید، ظاهر برنامه مرتب می‌ماند. امتحان کنید...

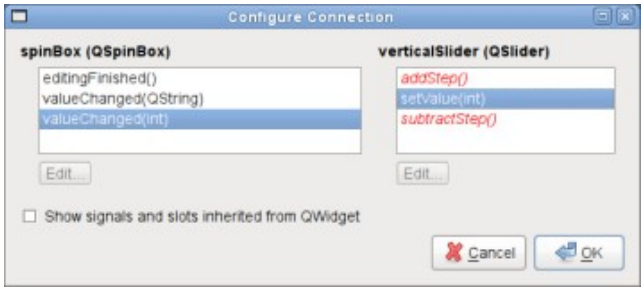
چیزی که در وهله اول با دیدن این فرم به ذهن کاربر می‌رسد، این است که با تغییردادن slider، باید مقدار عددی بالا نیز تغییر کند، و بالعکس! اما اگر با استفاده از کلید میانبر Ctrl+R پیش نمایش فرم را ببینید، این اتفاق نمی افتد! چه کنیم؟



راه حل ساده است:

با فشردن کلید میانبر F4 و یا انتخاب گزینه Edit Signals/Slots از منوی Edit طراح به حالت ویرایش سیگنال/اسلات می‌رود.

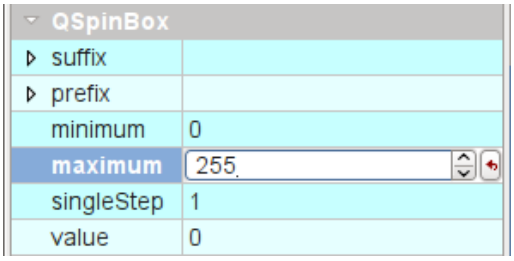
روی slider کلیک کرده و اشاره گر موس را به روی SpinBox بکشید، حالا پنجره تنظیم اتصال باز می‌شود که در عکس زیر مشاهده می‌کنید:



سیگنال و اسلات مورد نظر را در این پنجره انتخاب نمایید:

سیگنال ValueChanged از spinbox و اسلات setValue در slider. توجه داشته باشید که ورودی سیگنال و اسلات باید همسان باشند.

این عمل را برای دیگر رنگها هم تکرار نمایید.

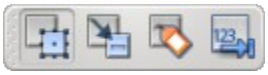


همانطور که می‌دانید، مقدار RGB ها بین ۰ تا ۲۵۵ می‌باشد. پس باید این محدودیت را در مقدار spinbox هم تعیین کنیم: روی اولین spinbox کلیک کنید و در قسمت ویرایشگر خصوصیات (Property Editor)، مقدار maximum را به ۲۵۵ تغییر دهید.

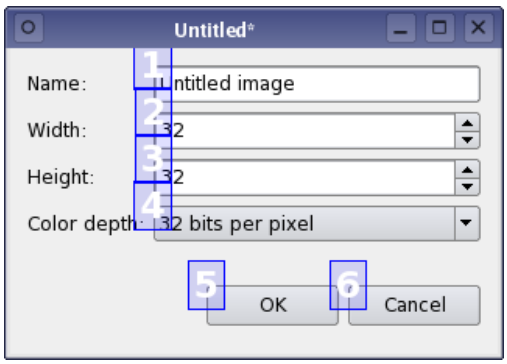
حالا با فشردن کلید میانبر Ctrl+R که قبلا هم اشاره شد، پیش نمایش فرم را ببینید!

حال با کشیدن slider مقدار عدد نیز تغییر می‌کند.

اما مشاهده خواهید کرد که با تغییر دادن مقدار spinbox، مقدار slider تغییری نمی کند! واضح است که ما این ارتباط را یکطرفه برقرار کردیم، برای داشتن این مسئله بطور دو طرفه، به حالت ویرایش سیگنالها و اسلاتها برگردید و تغییرات مورد نظر را اعمال نمایید.



برنامه طراح را در چهار حالت متفاوت می‌توانید استفاده نمایید، که با دو مورد آنها آشنا شدیم، حالت ویرایش که حالت پیش فرض برنامه است، و حالت ویرایش سیگنالها/اسلاتها. دو حالت دیگر برنامه عبارتند از: ویرایش buddy یا دوست، که برای مدیریت بهتر تمرکز صفحه‌کلید (Keyboard Focus) بکار می‌رود، احتمالا در ابتدای کار زیاد بکار شما نمی‌آید. و حالت دیگر: ویرایش ترتیب tab ویدجتها که برای تعیین ترتیبی که با کلید Tab میتوان روی ویدجتهای فرم حرکت کرد استفاده می‌شود، با کلیک کردن روی اعداد می‌توانید ترتیب آنرا مشخص کنید، طبق مثال زیر:



نحوه استفاده از خروجی برنامه‌ی طراح در برنامه:

بر خلاف برنامه های طراحای که در محیط ویندوز با آنها آشنا هستیم، برنامه‌ی طراح کیوت، کد C++ تولید نمی‌کند! بلکه فایلی با فرمت xml تولید می‌کند. توسعه دهندگان کیوت، ابزاری به نام uic یا UI Compiler را به همراه آن منتشر می‌کنند، که کار تولید کد C++ از فایل xml خروجی طراح را بر عهده دارد. دلایل فنی و فلسفی ای در این سیستم وجود دارد، که در بحث ما نمی گنجد، پس مستقیم سراغ نحوه استفاده از این خروجی xml در برنامه می‌رویم:

۲ اسلوب کلی در پروسس و استفاده از فایلهای ui وجود دارد:

- ❖ پردازش در زمان کامپایل (Compile time)
- ❖ پردازش در زمان اجرا (Run time)

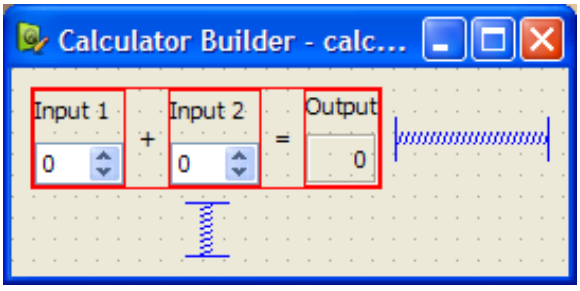
که هرکدام کاربردها و انعطاف پذیری خاص خود را دارند. اما ما بیشتر از پردازش در زمان کامپایل استفاده می‌کنیم، و آنرا باز می‌کنیم:

به سه روش می‌توان از کد تولید شده از یک فایل ui در زمان کامپایل استفاده کرد:

- روش مستقیم
- روش وراثت یگانه
- روش وراثت چندگانه

روش مستقیم:

برای توضیح اینکه چگونه از روش مستقیم استفاده نماییم از یک مثال ساده استفاده می‌کنیم: یک ماشین حساب ساده که تنها عمل جمع را انجام میدهد.



برنامه ما از دو فایل تشکیل می‌شود، یک فایل main.cpp که سورس برنامه را دارا می‌باشد. و یک فایل calculatorform.ui که فرم طراحی شده در برنامه طراح می‌باشد، فرم ما به شکل روبرو خواهد بود:

برای ساختن فایل اجرایی از سورس، از برنامه qmake استفاده می‌کنیم، پس به یک فایل .pro نیاز داریم:

```
TEMPLATE     = app
FORMS        = calculatorform.ui
SOURCES      = main.cpp
```

نکته مهم و جدید این فایل، خط دوم، *FORMS* است! که به qmake می‌گوید که کدام فایلها باید بوسیله uic پردازش گردند.

اینجا فایل calculatorform.ui مورد پردازش قرار گرفته و فایل ui_calculatorform.h که شامل کد C++ برای تولید فرم مورد نظر می‌باشد، تولید می‌شود.

جهت استفاده از فرم، ما باید فایل ui_calculatorform.h را در سورس برنامه وارد کنیم:

```
#include "ui_calculatorform.h"
```

نکته: در زمان کامپایل، اگر فایل uiای وجود داشته باشد، که خروجی آن (ui_NAME.h) در هیچکدام از فایلهایی که در قسمت SOURCES برای qmake بعنوان سورس فایلهای ما تعریف شده اند، استفاده نشده باشد، فایل .hای از آن تولید نخواهد شد! پس صرف معرفی کردن فایل ui به qmake طی تعریف فایل پروژه (.pro) برای تولید کد C++ از فرم کافی نمی باشد.

همانطور که ملاحظه میکنید، تابع main برنامه، ویدجت ماشین حساب را با ساختن یک QWidget که به عنوان میزبان فرم ما استفاده می‌شود، پیاده می‌کند:

```
int main(int argc, char *argv[])
{
  QApplication app(argc, argv);
  QWidget *widget = new QWidget;
  Ui::CalculatorForm ui;
  ui.setupUi(widget);

  widget->show();
  return app.exec();
}
```

در این مثال، کلاس Ui::CalculatorForm کلاس تولید شده از فایل ui می‌باشد.
این روش، یک راه سریع و ساده برای استفاده از اجزایی که خودشان همه چیز را به همراه دارند در برنامه می‌باشد.
اما بعد از اجرای این برنامه، متوجه خواهید شد که با تغییر مقدار spin boxها، نتیجه جمع تغییری نکرده، و در واقع برنامه کار نخواهد کرد. می‌توانید این قسمت را به کد بیافزایید؟ صددرصد باید بتوانید!
برای داشتن تسلط و قدرت بیشتر روی اجزای فرم و برنامه، می‌توانیم از روش وراثت یگانه استفاده نماییم:

روش وراثت یگانه:

در این روش، ما یک کلاس از کلاس QWidget مشتق می‌نماییم، و ظاهر گرافیکی را داخل سازنده کلاس تعریف و تنظیم می‌کنیم، و از این کلاس بعنوان میزبان فرم استفاده می‌کنیم. بدین طریق، می‌توانیم ارتباطات لازم را بین سیگنال و اسلاتهای اجزای گرافیکی، و دیگر اجزای مورد نظرمان در کلاس تعریف کنیم.
خوب، حالا ۲ فایل جدید به برنامه ما افزوده شده اند: calculatorform.h و calculatorform.cpp
پس دو خط زیر را به فایل .pro اضافه می‌نماییم:

```
HEADERS   = calculatorform.h
SOURCES  += calculatorform.cpp
```

و کلاس CalculatorForm به شکل زیر تعریف می‌گردد:

```
#include "ui_calculatorform.h"

class CalculatorForm : public QWidget
{
  Q_OBJECT

public:
  CalculatorForm(QWidget *parent = 0);

private slots:
  void on_inputSpinBox1_valueChanged(int value);
  void on_inputSpinBox2_valueChanged(int value);

private:
  Ui::CalculatorForm ui;
};
```

خصوصیت مهم کلاس ما، شی خصوصی (private) ای است که به نام ui تعریف شده است.

و سازنده کلاس به شکل زیر، در ابتدای ساخته شدن یک نمونه از کلاس، ظاهر گرافیکی را آماده استفاده می‌نماید:

```
CalculatorForm::CalculatorForm(QWidget *parent)
  : QWidget(parent)
{
  ui.setupUi(this);
}
```

از مهمترین فواید این روش، می‌توان به موارد زیر اشاره کرد:

- ✓ سادگی استفاده از وراثت، برای داشتن یک واسط بر پایه‌ی QWidget
- ✓ کپسوله سازی و مخفی سازی متغیر ui که مربوط به ظاهر گرافیکی می‌شود.
- ✓ می توان از این روش برای ساختن ظاهرهای متفاوت و متنوع (که هرکدام فضای خود را دارا باشند) داخل یک ویدجت استفاده کرد.
- ✓ همچنین می‌تواند جهت ساختن tabهای گوناگون از فرمهای موجود استفاده شود.

روش وراثت چندگانه:

یکی از قابلیتها و قدرتهای زبان C++ امکان پیاده سازی وراثت چندگانه است، به این شکل که یک کلاس می‌تواند ویژگیهایی را از چندین کلاس متفاوت بطور همزمان به ارث برد.

اما پیاده سازی ما بدین صورت خواهد بود که کلاس CalculatorForm بطور همزمان هم از یک ویدجت(بطور مثال QWidget) ارث می‌برد، هم از Ui::CalculatorForm

پس سورس کد فایل calculatorform.h به این شکل خواهد شد:

```
#include "ui_calculatorform.h"

class CalculatorForm : public QWidget, private Ui::CalculatorForm
{
  Q_OBJECT

public:
  CalculatorForm(QWidget *parent = 0);

private slots:
  void on_inputSpinBox1_valueChanged(int value);
  void on_inputSpinBox2_valueChanged(int value);
};
```

همانطور که ملاحظه می‌نمایید دیگر متغییری بنام ui تعریف نشده است، چرا که اینجا از داخل خود کلاس امکان دسترسی به اشیاء داخل فرم را داریم.
و وراثت کلاس Ui::CalculatorForm را بطور خصوصی(private) تعریف کرده ایم، تا مطمئن باشیم که اشیای تعریف شده در فایل ui، در کلاس خصوصی خواهند بود. البته که می‌توانیم بصورت عمومی (public) و یا محافظت شده(protected) ارث بری نماییم، همانطور که می‌توانستیم متغیر ui را عمومی یا محافظت شده تعریف نماییم.

و دست آخر سازنده کلاس ما به این شکل خواهد بود:

```
CalculatorForm::CalculatorForm(QWidget *parent)
  : QWidget(parent)
{
  setupUi(this);
}
```

جهت کسب اطلاعات بیشتر در مورد روش استفاده از فایلهای ui در زمان اجرا به آدرس زیر مراجعه نمایید.
http://doc.qt.nokia.com/latest/designer-using-a-ui-file.html

فصل ششم: کار با پایگاه داده ها در کیوت

پایگاه داده یکی از بخشهای مهم در حتی برنامه های ساده است. در حالی که خیلی از خواننده ها فکر می‌کنند پایگاه داده ها تنها در برنامه های بزرگ و وبسایت ها مورد استفاده قرار می‌گیرد، امروزه برنامه های ساده و کوچک هم می‌توانند به سادگی و برای سادگی و قابل فهم بودن بیشتر برنامه از پایگاه داده ها استفاده کنند، حتی برای نگهداری تنظیمات برنامه.

اما کیوت در این زمینه چه چیزی به ما می‌دهد؟

یک سیستم بدون وابستگی به سیستم عامل و حتی بدون وابستگی به پایگاه داده. از این لحاظ که کدی که شما می‌نویسید می‌تواند با پایگاه داده های مختلف از جمله، MySQL, Oracle, SQLite, PostgreSQL, SyBase, DB2, Interbase و ODBC کار کند و حتی کیوت این قابلیت را به برنامه نویسان می‌دهد تا بتوانند پشتیبانی از یک پایگاه داده‌ی جدید را نیز به آن بیافزایند. تنها تفاوت بین پایگاه داده های مختلف، تفاوت بین SQLهایی که آنها پشتیبانی میکنند است.

در این مبحث نحوه‌ی استفاده از پایگاه داده های MySQL و SQLite در برنامه Qt را بررسی می‌کنیم. MySQL برای برنامه های بزرگ، و SQLite زمانی که استفاده از پایگاه داده مفید است اما نیازی به یک پایگاه داده خیلی پیشرفته نداریم.

پیشفرض ما در این بحث این است که خواننده حداقل با مفهوم پایگاه داده‌ها و زبان عمومی SQL آشنایی دارد، و با یک پایگاه داده قبلا کار کرده است.

کلاسهای کیوت برای مدیریت و برقراری ارتباط با پایگاه داده ها را می‌توان به سه گروه تقسیم کرد.

لایه‌ی اول بر پایه‌ی یکسری driver پایگاه داده ها می‌چرخد. (در اینجا به پلاگینی که پشتیبانی از یک پایگاه داده را برای کیوت فراهم می‌کند، driver آن پایگاه داده می‌گویند.)

لایه‌ی دوم ارتباط با پایگاه داده ها و پرس و جوها (queries) و نتایج آنرا مدیریت می‌کند. این لایه برپایه‌ی لایه‌ی اول می‌باشد، زیرا برای داشتن ارتباط با یک پایگاه داده به driver آن نیازمندیم.

لایه سوم، که لایه‌ی واسط کاربری نامیده میشود، یکسری ماژولها و ابزارهایی برای استفاده با چارچوب مدل کیوت (Qt's Model View Framework) فراهم می‌کند.

برقراری ارتباط

هر ارتباطی با پایگاه داده بوسیله‌ی یک شیئ QSqlDatabase نمایش داده می‌شود. و ارتباط توسط driver برقرار می‌گردد. بعد از انتخاب driver مربوطه، شما می‌توانید مشخصات مربوط به ارتباط و دیتابیس را تنظیم کنید. و بعد از تنظیمات، شما باید پایگاه داده را باز کنید، تا بتوانید از آن استفاده کنید.

برای جلوگیری از اینکه برای هر کاری، این شیئ دیتابیس را به توابع پاس بدهیم، کل ماژول QtSql یک ارتباط پیشفرض(default connection) دارد. و از جایی که در هر زمانی تنها با یک پایگاه داده کار می‌کنیم، کلیه کلاسهای مربوطه از همان ارتباط پیشفرض استفاده خواهند کرد.

کد زیر برقراری ارتباط با یک پایگاه داده MySQL را نشان می‌دهد. پروسه‌ی انجام آن ساده است، ابتدا یک ارتباط با استفاده از درایور QMYSQL و از طریق تابع ایستا(static)ی QSqlDatabase::addDatabase می‌افزاییم. از جایی که تنها اسم driver را به آن پاس می‌دهیم، و نامی برای ارتباط در نظر نمی گیریم، این همان ارتباط پیشفرض خواهد بود.

سپس شیئ QSqlDatabase نتیجه، تنظیم می‌گردد. مشخصات hostName, databaseName, userName, و passwordکه کارشان از نام آنها مشخص است، تنظیم می‌گردند.

سپس ارتباط را بوسیله تابع open برای استفاده باز می‌کنیم، اگر خروجی این تابع false بود، یعنی ارتباط برقرار نشده است. دلیل این مشکل، بوسیله‌ی یک شیئ QSqlError که از طریق تابع lastError قابل دسترسی است، قابل استفاده است. و اگر خروجی تابع open مقدار true باشد، ارتباط برقرار شده و آماده استفاده است.

نکته: مشخصاتی که در زمان برقراری ارتباط استفاده می‌شوند، hostName, databaseName, userName, password, port و connectOptions و مقداری که به این توابع می‌دهیم، بسته به درایور مورد استفاده، متفاوت خواهد بود.

```
QSqlDatabase db = QSqlDatabase::addDatabase( "QMYSQL" );
db.setHostName( "localhost" );
db.setDatabaseName( "qtbook" );
db.setUserName( "user" );
db.setPassword( "password" );
if( !db.open() )
{
   qDebug() << db.lastError();
   qFatal( "Failed to connect." );
}
```

و کد زیر نحوه برقراری ارتباط با یک پایگاه داده SQLite را از طریق درایور QSQLITE نشان می‌دهد.

SQLite با MySQL تفاوتهای بنیادینی دارد، زیرا چیزی بنام سرور و لاگین کردن ندارد، پس username و password نیز ندارد. تنها لازم است آدرس فایلی که پایگاه داده را نگه می‌دارد به عنوان databaseName به برنامه بدهیم، پس فایل مورد نظر باز و یا ایجاد می‌گردد (زمان فراخوانی تابع open).

```
QSqlDatabase db = QSqlDatabase::addDatabase( "QSQLITE" );
db.setDatabaseName( "testdatabase.db" );
if( !db.open() )
{
   qDebug() << db.lastError();
   qFatal( "Failed to connect." );
}
```

یکی از قابلیتهای خوب SQLite این است که دیتابیس می‌تواند در حافظه‌ی اصلی سیستم (Ram) ایجاد گردد و مورد استفاده قرار گیرد. اینگونه سرعت کار خیلی بیشتر خواهد بود، زیرا هزینه‌ی نوشتن و خواندن از دیسک را نداریم. البته در صورتی که بخواهیم بعد از اتمام برنامه، اطلاعات را داشته باشیم، این روش جوابگو نخواهد بود. برای اینکار فایل دیتابیس را :memory: تعیین می‌کنیم:

```
db.setDatabaseName( ":memory:" );
```

زمانی که یک شیئ QSqlDatabase که یک ارتباط را نشان می‌دهد، دیگر استفاده نمی شود، می‌توان آنرا بوسیله تابع close بست! البته همه‌ی ارتباطاتی که باز مانده باشند، در پایان کار بطور خودکار توسط تابع خراب کننده(destructor) شیئ QSqlDatabase بسته خواهند شد.

گرفتن اطلاعات از پایگاه داده ها

زمانی که یک کوئری SQL برای اجرا به پایگاه داده می‌دهیم، از یک شیئ QSqlQuery هم برای کوئری و هم برای نگهداری نتیجه آن استفاده می‌کنیم. با یک دستور SELECT ساده آغاز می‌کنیم:

کد زیر اجرای یک کوئری را نشان می‌دهد، کوئری SQL را به دستور exec شیئ QSqlQuery پاس می‌دهیم، اگر اجرا با شکست روبرو گردد، تابع exec مقدار false برمی گرداند، در این شرایط تابع lastError() اطلاعات بیشتری در مورد مشکل خواهد داد. لزوما مشکل از کوئری نخواهد بود، قطع شدن ارتباط با سرور و خطاهایی از این قبیل نیز، ممکن است در این قسمت اتفاق بیفتد.

```
if( !qry.exec( "SELECT firstname, lastname FROM names "
               "WHERE lastname = 'Roe' ORDER BY firstname" ) )
      qDebug() << qry.lastError();
```

اگر کوئری بدون اشکال اجرا شد، آنگاه وقت آن رسیده که از نتایج اجرای آن استفاده کنیم.

همانطور که می‌دانید، نتیجه اجرای یک SELECT همیشه یک جدول است. (البته این جدول میتواند تنها یک ردیف و یک ستون داشته باشد.)

کد زیر نشان می‌دهد چگونه نتایج را از کوئری می‌گیریم:

```
QSqlRecord rec = qry.record();
int cols = rec.count();
QString temp;
for( int c=0; c<cols; c++ )
      temp += rec.fieldName(c) + ((c<cols-1)?"\t":"");
qDebug() << temp;
while( qry.next() )
{
      temp = "";
      for( int c=0; c<cols; c++ )
            temp += qry.value(c).toString() + ((c<cols-1)?"\t":"");
      qDebug() << temp;
}
```

اول یک QSqlRecord می‌گیریم، هر رکورد نشان دهنده‌ی یک ردیف در خروجی کوئری است. و با تابع count میتوان تعداد ستونهای خروجی را فهمید. اسم ستونهای نتیجه را با استفاده از تابع fieldName(int) می‌توانید بدست آورید. با این دو تابع متنی با نام ستونها در for اول ساخته می‌شود.

برای دسترسی به اولین ردیف از خروجی، باید تابع next() را فراخوانی کنید، در واقع وقتی کوئری اجرا شد، شیئ QSqlQuery ما هنوز به هیچ کدام از ردیفهای خروجی اشاره نمیکند، و مقدار آن NULL است که با استفاده از تابع isValid میتوان این را فهمید. ابتدا باید تابع next() فراخوانی شود، حالا شیئ QSqlQuery به اولین ردیف خروجی اشاره می‌کند، و میتوان از آن استفاده کرد.

در اینجا اجرای while که شرطش فراخوانی next() است، باعث میشود این تابع اجرا گردد، و اگر ردیفی در خروجی ما وجود دارد، وارد حلقه گردد، و الا وارد حلقه نخواهد شد.

نکته: اگر خروجی اجرای کوئری خالی باشد، یعنی هیچ ردیفی با شرطی که در WHERE گذاردیم همخوان نباشد، کوئری به درستی اجرا می‌گردد، ولی خروجی آن خالی است. در واقع خطایی اتفاق نیافتاده است.

نکته: در صورتی که بعد از اجرا تابع next() مقداری در خروجی برای خواندن مانده باشد، این تابع true و در صورتی که هیچ ردیفی نباشد، مقدار false برمی گرداند.

نکته: تابع next فقط با کوئری SELECT کار می‌کند. البته که زمانی که کوئری ما SELECT نیست، چیزی برای گرفتن هم وجود ندارد. با استفاده از تابع isSelect میتوان فهمید که آیا کوئری اجرا شده SELECT بوده یا خیر!؟

برای هر ردیف، مقدار هر ستون با استفاده از تابع value(int) که ورودی آن شماره‌ی ستون است، گرفته می‌شود. خروجی تابع value یک شیئ از نوع QVariant است. پس ما باید آنرا به نوع داده‌ی مورد نظرمان تبدیل کنیم، این کار با استفاده از تابع toString برای تبدیل به یک رشته، قابل انجام است. کلاس QVariant توابعی برای تبدیل به اکثر انواع داده ای دارد، مثل toBool ، toDouble ، toInt و ...

در مثال قبل، کوئری را در قالب یک رشته ثابت به QSqlQuery پاس کردیم، در حالی که در دنیای واقعی اکثر اوقات مقادیری از کوئری هستند که از کاربر گرفته می‌شود و باید به کوئری اضافه شوند، بعنوان مثال، اگر در کد قبل، میخواستیم مقدار 'lastname = 'Roe را از کاربر گرفته و به کوئری پاس بدهیم، چه!؟

خوب ساده ترین راه حلی که به ذهن می‌رسد، افزودن آن به متن کوئری است مثل زیر:

```
  queryText = "SELECT firstname, lastname FROM names WHERE lastname = ' " +
userSuppliedText + " ' ORDER BY firstname"
```

و سپس دادن متغیر queryText به تابع exec

اما این راه، راه بهینه و درستی نیست، چرا که ممکن است اسمی که کاربر وارد می‌کند، کاراکتر ' داشته باشد! آنگاه، پایگاه داده در اجرای این کوئری به مشکل برخواهد خورد.

راه حل چسباندن (bind) مقدار، پیش از اجرا به کوئری است.

```
qry.prepare( "INSERT INTO names (id, firstname, lastname) "
             "VALUES (:id, :firstname, :lastname)" );
qry.bindValue( ":id", 9 );
qry.bindValue( ":firstname", "Ralph" );
qry.bindValue( ":lastname", "Roe" );
if( !qry.exec() )
  qDebug() << qry.lastError();
```

این کد، نحوه‌ی انجام اینکار را برای یک کوئری INSERT نشان می‌دهد. آماده سازی کوئری که یک مرحله‌ی اختیاری است، ممکن است در بعضی پایگاه داده ها املا (syntax)کوئری را نیز چک کند. اگر اشکالی در کوئری باشد، تابع prepare مقدار false برخواهد گرداند. و حتی اگر در این مرحله کوئری تست گردد، باز هم تابع exec ممکن است با خطا مواجه شود، خطاهای خود پایگاه داده و ارتباط با آن.

کد بالا، نشان می‌دهد که در زمان آماده سازی(prepare) کوئری، ما بجای مقادیر از متغیرهایی که با دو نقطه(:) آغاز می‌شوند، استفاده می‌کنیم. و بعد از آماده سازی، مقدار آنها را بوسیله تابع bindValue(QString,QVariant) تعیین می‌کنیم.

نکته: شما می‌توانید از علامت سوال (?) بجای متغیرها استفاده کنید، و بعد از آن مقادیر را با استفاده از تابع addBindValue(QVariant) به ترتیب از چپ به راست تنظیم نمایید. البته این روش سریعتر و ساده تر ولی احتمال خطای برنامه نویس در آن بیشتر است.

برقراری چند ارتباط بطور همزمان

اگر شما نیاز به برقراری ارتباط با چند پایگاه داده بطور همزمان دارید، باید برای ارتباطات نام تعیین کنید. (در صورت عدم تعیین نام، از ارتباط پیش فرض استفاده می‌شود.) در صورتی که یک پایگاه داده با نامی تکراری بیافزایید، ارتباطی که قبلا برقرار شده بود با این نام، از دست می‌رود، و ارتباط جدید جایگزین آن میشود، این اتفاق برای ارتباط پیشفرض نیز می‌افتد.

برای تعیین نام، زمان فراخوانی تابع QSqlDatabase::addDatabase نام ارتباط را بعنوان آرگومان دوم به آن بدهید:

```
QSqlDatabase::addDatabase("QSQLITE","CONNECTION_NAME");
```

و در زمان ساختن یک QSqlQuery می‌توانید یک شیئ QSqlDatabase به آن بدهید، که در صورتی که میخواهید از این دیتابیس، بجای دیتابیس پیشفرض استفاده کند، اینکار ضروری است.

دسترسی به دیتابیسهایی که در حال حاضر آنها را بوسیله‌ی تابع QSqlDatabase::addDatabase به برنامه افزوده ایم، بوسیله تابع استاتیک QSqlDatabase::database(QString) امکان پذیر است، که ورودی آن همان CONNECTION_NAME است که هنگام تعریف و افزودن ارتباط تعیین کردیم.

مثالی برای یک برنامه کامل

برای اینکه همه‌ی یافته های خود را کنار هم بگذاریم، بیایید یک برنامه‌ی ساده که از یک دیتابیس SQLite برای نگهداری اطلاعات استفاده می‌کند، بنویسیم:

برنامه‌ی ما یک کلکسیون عکس است، که میتوانیم:

- به آن عکس اضافه کنیم.
- عکسها را برچسب (tag) بزنیم.
- و عکسهایی با یک tag خاص را نمایش بدهیم.

عکسها و tagها را در یک پایگاه داده نگهداری می‌کنیم.

برنامه‌ی ما از یک پنجره ساده تشکیل شده است،(عکس زیر را ببینید)

tagها سمت راست نمایش داده می‌شوند. و تعداد عکسهایی که با آن tagها موجود است در دیتابیس، پایین لیست در یک برچسب نمایش داده می‌شود. نیمه‌ی چپ پنجره برای نمایش عکس استفاده می‌شود. و دکمه های برنامه که برای جابجایی بین عکسها و افزودن عکس و tag استفاده می‌شوند، در پایین پنجره قرار دارند.

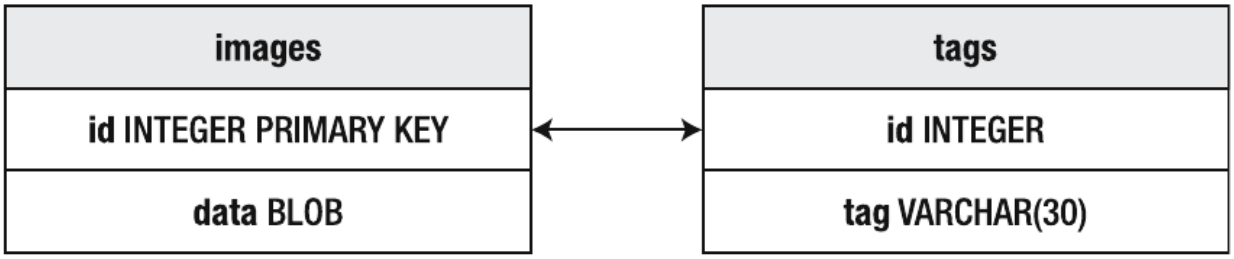
همانطور که در تصویر می‌بینید، برنامه‌ی ما دو بخش را شامل می‌شود، افزودن به دیتابیس (که شامل افزودن عکس و tag می‌شود) و نمایش محتوای دیتابیس، که شامل نمایش عکس و نمایش tagها می‌شود. اما شما

می‌توانید بعد از اتمام این بحث، بخشهای بیشتری نیز برای تمرین به آن بیافزایید، مثل حذف کردن، ویرایش کردن و ...

پایگاه داده ای که برای پیاده سازی این برنامه استفاده می‌کنیم، از دو جدول تشکیل می‌گردد، یکی برای نگهداری عکس ها و دیگری برای نگهداری tag ها. (دو جدول images و tags همانطور که در عکس بعد می‌بینید)

در جدول images هر عکس در یک ردیف نگهداری می‌گردد. و هر ردیف یک مقدار int که id آن است، و یک مقدار از نوع BLOB که data می‌نامیم و عکس در آن نگهداری می‌گردد.

جدول tags هم دو ستون دارد، id و tag . البته id در اصل ImageId می‌باشد، یعنی شناسه‌ی عکسی که این tag به آن مرتبط است.



ساختار برنامه

برنامه از دو بخش اصلی تشکیل شده است، بخش واسط کاربر، و بخش ارتباط با دیتابیس. بخش واسط کاربری از بخش ارتباط برای برقراری ارتباط با پایگاه داده استفاده می‌کند. و این بخش در کلاس ImageDialog پیاده سازی می‌گردد. بخش ارتباط با دیتابیس هم در کلاس ImageCollection پیاده سازی می‌گردد.

با جدا کردن کد ارتباط با پایگاه داده، شما از وجود کوئریهای SQL در همه جای برنامه جلوگیری می‌نمایید، البته دلایل دیگری نیز برای این عمل وجود دارند. اول از همه، آن قسمت از کد می‌تواند بطور جداگانه مورد آزمایش قرار گیرد، و ایرادات کوئریها و ارتباطات برطرف گردند. اگر ما تبدیلات بین نوع داده های پایگاه داده و Qt را در یک مکان انجام دهیم کد ساده تر و خواناتر خواهد بود. و حتی اگر شما زمانی بخواهید موتور پایگاه داده را عوض کنید، بعنوان مثال از MySQL و یا PostgreSQL استفاده نمایید، انجام اینکار به سادگی تغییر تنها یک کلاس یا یک بخش از کد خواهد بود.

واسط کاربری

همانطور که قبلتر گفتیم، واسط کاربری برنامه در کلاس ImageDialog پیاده می‌گردد، تعریف کلاس که در کد زیر قابل مشاهده و بررسی است، از یک سازنده بعلاوه چند اسلات تشکیل شده است، که هر اسلات یکی از کارهایی که کاربر با برنامه می‌تواند انجام دهد را مدیریت می‌کند.

کاربر چه کارهایی می‌تواند بکند!؟ با بررسی تعریف ما از برنامه و دیدن دوباره ظاهر برنامه، میتوان جوابهایی برای این سوال پیدا کرد:

- حرکت کردن بین عکسها: nextClicked و previousClicked
- تغییردادن tagهای انتخاب شده: tagsChanged
- افزودن یک عکس جدید: addImageClicked
- افزودن یک tag جدید: addTagClicked

به این لیست می‌توان کارهای عمومی ای که کاربر میخواهد بکند، از جمله خارج شدن از برنامه را نیز افزود.

```
class ImageDialog : public QDialog
{
Q_OBJECT
public:
  ImageDialog();
private slots:
  void nextClicked();
  void previousClicked();
  void tagsChanged();
  void addImageClicked();
  void addTagClicked();
  ...
};
```

ادامه‌ی تعریف کلاس، به ما می‌گوید که کلاس چگونه کار می‌کند! کد زیر را ببینید، این قسمت با تعریف چهار تابع پشتیبانی، شروع می‌شود: selectedTags, updateImages, updateTags, و updateCurrentImage. به زودی شما آنها را تک به تک خواهید دید.

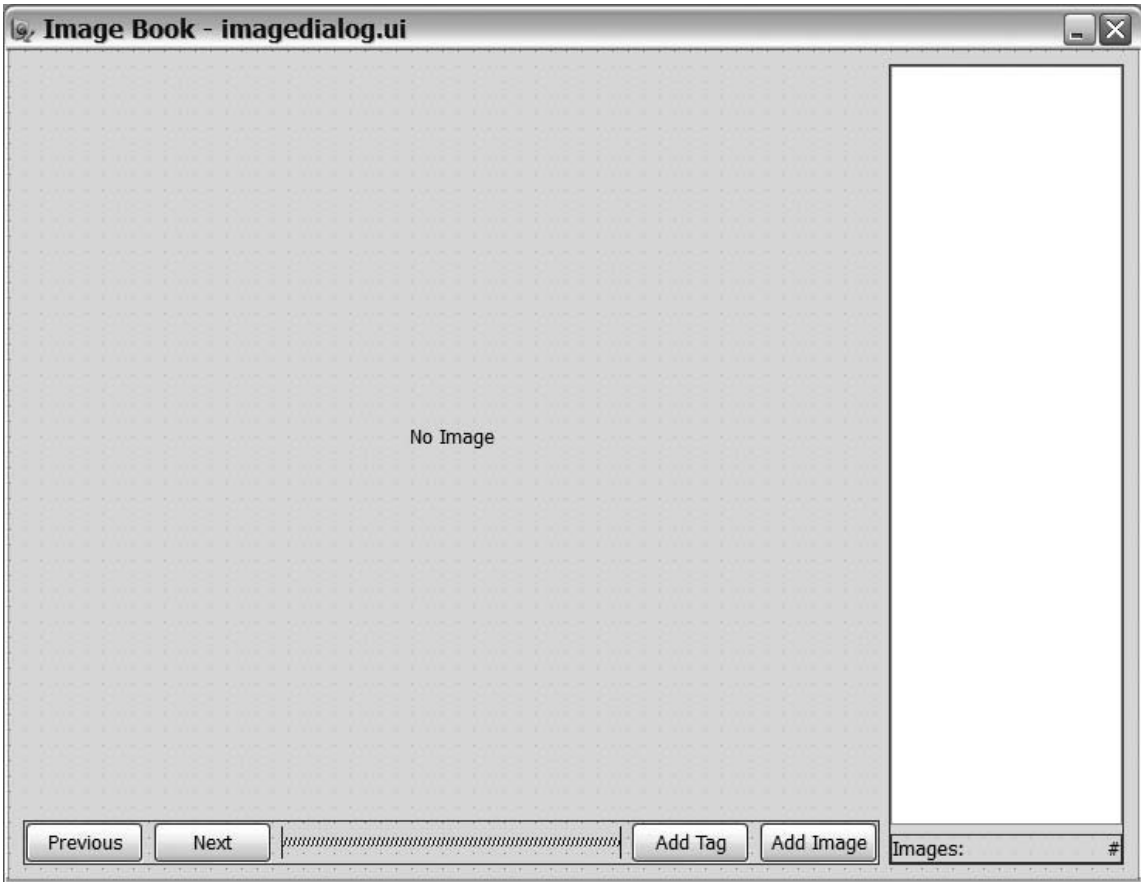
بعد از توابع، واسط دیداری برنامه را که با استفاده از برنامه طراح کیوت آماده شده است می‌بینید که با نام ui قبل از متغیرهایی که برای نگهداری اطلاعات در مورد عکسها استفاده می‌شوند، قرار دارد. متغیر currentImage بعنوان یک اندیس به لیست عکسها که در imageIds است، استفاده می‌شود که در هر لحظه نشان میدهد کدام عکس فعال است.

و در پایان متغیر images که یک نمونه (instance) از کلاس ImageCollection است، که گفتیم ارتباط با پایگاه داده را مدیریت میکند، خواهیم داشت.

```
class ImageDialog : public QDialog
{
...
private:
 QStringList selectedTags();

 void updateImages();
 void updateTags();
 void updateCurrentImage();

 Ui::ImageDialog ui;
 QList<int> imageIds;
 int currentImage;

 ImageCollection images;
};
```

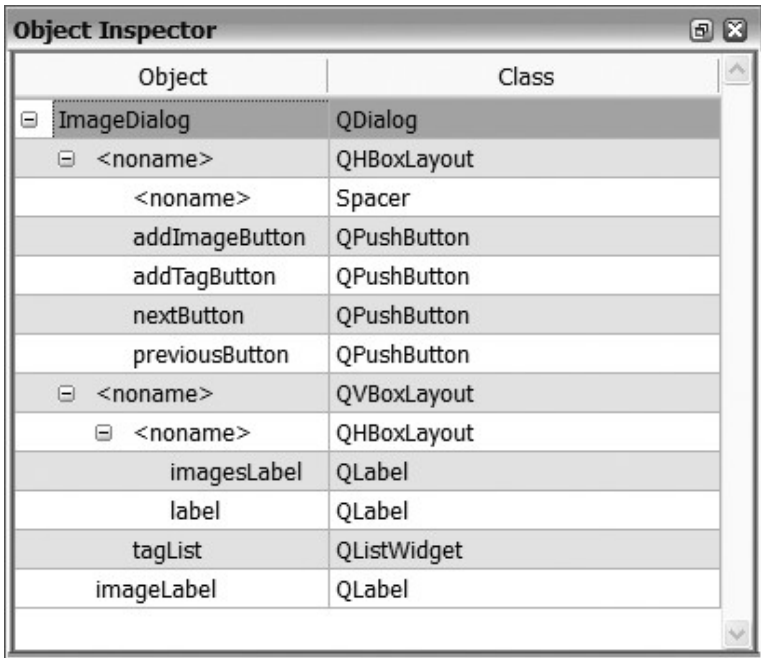
ویدجتها و اسلاتها

ImageDialog با استفاده از برنامه طراح کیوت (Qt Designer) ساخته شده است، میتوانید عکس آنرا ببینید:



غیر از نام ویدجتها که برای دستیابی به آنها مهم است و ما آنها را تعیین کرده ایم، تنها چیزی که از خصوصیات ویدجتها تغییر کرده است، مشخصه‌ی SelectionMode از QListWidget است که به MultiSelection تغییر کرده است.

عکس زیر سلسله مراتب ویدجتها در پنجره را نشان می‌دهد، شما میتوانید نام و نوع اشیاء را نیز ببینید:



حال بیایید تا نگاهی به کد کلاس ImageDialog بیاندازیم. از سازنده کلاس شروع می‌کنیم، کدی که قبل از اینکه کاربر برنامه را ببیند اجرا می‌شود:

```
ImageDialog::ImageDialog()
{
  ui.setupUi( this );

  currentImage = -1;

  updateTags();
  updateImages();

  connect( ui.previousButton, SIGNAL(clicked()),
           this, SLOT(previousClicked()) );
  connect( ui.nextButton, SIGNAL(clicked()),
           this, SLOT(nextClicked()) );
  connect( ui.addTagButton, SIGNAL(clicked()),
           this, SLOT(addTagClicked()) );
  connect( ui.addImageButton, SIGNAL(clicked()),
           this, SLOT(addImageClicked()) );
  connect( ui.tagList, SIGNAL(itemSelectionChanged()),
           this, SLOT(tagsChanged()) );
}
```

کد با تنظیم واسطی که توسط برنامه طراح ساخته شده است شروع می‌شود. مقدار currentImage را به یک مقدار غیر معقول تنظیم میکند، تا مطمئن باشیم قبل از گرفتن لیست تگها و عکسها از دیتابیس هیچ عکسی نمایش داده نمی شود. سپس تگها و عکسها را به روز می‌کنیم و سیگنال کلیک شدن هریک از دکمه های برنامه را به اسلات مربوطه متصل می‌نماییم.

به یاد داشته باشیم تابع updateCurrentImage دکمه های Previous ، Next و Add Tags را غیرفعال میکند. این تابع از تابع updateImages که اینجا در سازنده فراخوانی شده است، فراخوانی می‌گردد. و این بدین معنا است که اگر این دکمه ها کلیک شدند، یعنی ما عکسی در currentImage داریم و مقدار آن درست است.

با بررسی اسلاتها متوجه میشویم که سه تا از آنها ساده هستند:

```
void ImageDialog::nextClicked()
{
  currentImage = (currentImage+1) % imageIds.count();
  updateCurrentImage();
}

void ImageDialog::previousClicked()
{
  currentImage--;
  if( currentImage == -1 )
       currentImage = imageIds.count() - 1;
  updateCurrentImage();
}
```

همانطور که پیشتر بحث شد، متغیر currentImage همچون اشاره گری، همیشه ما را به عکسی که در حال حاضر نمایش داده می‌شود، راهنمایی میکند. پس next و previous به سادگی افزودن و کاستن از مقدار این متغیر است، البته که باید توجه داشت که این مقدار از بازه‌ی تعداد عکسها خارج نشود.

```
void ImageDialog::tagsChanged()
{
  updateImages();
}
```

زمانی که تگهای انتخاب شده تغییر میکنند، ما باید لیست عکسهایی که به این تگها تعلق دارند را بگیریم، تابع updateImages اینکار را انجام خواهد داد.

کد مربوط به اسلات بعدی به شرح زیر است:

```
void ImageDialog::addTagClicked()
{
    bool ok;
    QString tag = QInputDialog::getText(this, tr("Image Book"), tr("Tag:"),
                                        QLineEdit::Normal, QString(), &ok );
    if( ok )
    {
      tag = tag.toLower();
      QRegExp re( "[a-z]+" );
      if( re.exactMatch(tag))
      {
         QMessageBox::warning( this, tr("Image Book"),
                           tr("This is not a valid tag. "
                           "Tags consists of lower case characters a-z.") );
         return;
      }
      images.addTag( imageIds[ currentImage ], tag );
      updateTags();
    }
}
```

اسلات زمانی که کاربر بخواهد یک تگ جدید بیافزاید، فراخوانی خواهد شد.

اسلات با نشان دادن یک QInputDialog از کاربر میخواهد که یک تگ وارد کند، در ادامه این تگ را به حروف کوچک تبدیل می‌کند و چک می‌کند که کاراکتری غیر از a تا z در آن نباشد. که این قسمت با یک عبارت منطقی (regular expression) انجام می‌گیرد.

زمانی که مطمئن شدیم که تگ وارد شده، قابل قبول است، از شیئ ImageCollection می‌خواهیم که این تگ را به تگهای این عکس بیافزاید. و بعد از آن که تگ افزوده شد، لازم است تا لیست تگها را با فراخوانی updateTags به روز کنیم.

نکته: در کد تابع addTagClicked متونی که در واسط برنامه دیده خواهند شد مثل Image Book که عنوان پنجره‌های نمایش‌داده شده است، و یا پیغام خطا، را در تابع tr به برنامه پاس دادیم! اینکار باعث می‌شود به سادگی ترجمه‌کردن این متون، برنامه‌ی ما به زبانهای دیگر ترجمه گردد. (tr از کلمه translate گرفته شده‌است)

جهت کسب اطلاعات بیشتر به مستندات این تابع در آدرس زیر مراجعه نمایید:

http://doc.qt.nokia.com/latest/qobject.html#tr

اسلات باقیمانده addImageClicked است، که در ادامه کد و بررسی آنرا می‌بینید:

```
void ImageDialog::addImageClicked()
{
  QString filename = QfileDialog::getOpenFileName( this, tr("Open file"),
                                                    QDir::currentPath(),
                                                    tr("PNG Images (*.png)") );
  if( !filename.isNull() )
  {
    QImage image( filename );
    if( image.isNull() )
    {
      QMessageBox::warning( this, tr("Image Book"),
      tr("Failed to open the file '%1'").arg( filename ) );
      return;
    }
    images.addImage( image, selectedTags() );
    updateImages();
  }
}
```

در اینجا تعیین کرده‌ایم که تنها عکسهای png برای ما قابل قبول هستند. زمانی که یک عکس انتخاب شد، آنرا بارگذاری (load) می‌کنیم، اگر با موفقیت بارگذاری شد، عکس به ImageCollection اضافه می‌شود، بهمراه تگهایی که در حال حاضر انتخاب شده اند. برای گرفتن تگها، از تابع selectedTags استفاده کردیم. بعد از اضافه شدن عکس به پایگاه داده ها باید، لیست id عکسها بروز شود.

همانطور که دیدید، اسلاتها ساده و جالب بودند. آنها گهگاه چک میکنند که مقداری که کاربر وارد کرده است، قابل قبول است، قبل از آنکه آنرا به شیئ ImageCollection بدهد. زمانی که چیزی باید به روز شود، تابع پشتیبانی مربوطه را فراخوانی می‌کند. و ...

توابع پشتیبانی

تابع selectedTags که توسط اسلاتها و توابع پشتیبانی استفاده شده است، و در هر لحظه تگهای انتخاب شده را در یک QStringList بر می‌گرداند، روی Item های QListWidget حرکت می‌کند، و هرکدام که انتخاب شده‌اند، را به لیست اضافه می‌کند، کد زیر:

```
QStringList ImageDialog::selectedTags()
{
    QStringList result;
    foreach( QListWidgetItem *item, ui.tagList->selectedItems() )
      result << item->text();
    return result;
}
```

اولین تابع پشتیبانی که در سازنده کلاس فراخوانی شد، updateTags بود، که لیست تگها را بروز می‌کند، بدون تغییر در وضعیت انتخاب شده ها. (یعنی آنها که انتخاب شده هستند، بعد از بروز رسانی همچنان انتخاب شده میمانند.)

بدین نحو که ابتدا لیست تگهای انتخاب شده را از QListWidget میگیرد، سپس لیست را از ImageCollection گرفته، لیست را به روز می‌کند، و در نهایت دوباره تگهای انتخاب شده را انتخاب می‌کند:

```
void ImageDialog::updateTags()
{
  QStringList selection = selectedTags();
  QStringList tags = images.getTags();
  ui.tagList->clear();
  ui.tagList->addItems( tags );
  for( int i=0; i<ui.tagList->count(); ++i )
    if( selection.contains( ui.tagList->item(i)->text() ) )
      ui.tagList->item(i)->setSelected( true );
}
```

پس از آپدیت کردن لیست تگها، سازنده کلاس لیست عکسها را بروز می‌کند، با فراخوانی updateImages بروزرسانی لیست imageIds بر عهده این تابع است، همینطور در صورت وجود عکس کنونی در لیست جدید، مواظب است که عکس کنونی همچنان نمایش داده شود.

```
void ImageDialog::updateImages()
{
    int id;
    if( currentImage != -1 )
      id = imageIds[ currentImage ];
    else
      id = -1;
    imageIds = images.getIds( selectedTags() );
    currentImage = imageIds.indexOf( id );
    if( currentImage == -1 && !imageIds.isEmpty() )
      currentImage = 0;
    ui.imagesLabel->setText( QString::number( imageIds.count() ) );
    updateCurrentImage();
}
```

همانطور که می‌بینید، در ابتدا idی عکسی که در حال حاضر نمایش داده می‌شود را نگه میدارد، سپس لیست جدید عکسها را بر پایه تگهای منتخب کنونی از کلکسیون عکسها(ImageCollection) گرفته، عکس کنونی را آپدیت می‌کند(updateCurrentImage).

تابع updateCurrentImage که در ادامه می‌آید، ابتدا چک می‌کند که آیا هیچ currentImageی وجود دارد!؟ در صورت وجود، آنرا از ImageCollection گرفته، روی برچسب نمایش می‌دهد، و در صورت نبود آن، متن «No Image» را نمایش می‌دهد و دکمه های Next و Previous و Add Tag را غیرفعال می‌کند.

```
void ImageDialog::updateCurrentImage()
{
  if( currentImage == -1 )
  {
    ui.imageLabel->setPixmap( QPixmap() );
    ui.imageLabel->setText( tr("No Image") );
    ui.addTagButton->setEnabled( false );
    ui.nextButton->setEnabled( false );
    ui.previousButton->setEnabled( false );
  }
  else
  {
    ui.imageLabel->setPixmap(
    QPixmap::fromImage(
    images.getImage( imageIds[ currentImage ] ) ) );
    ui.imageLabel->clear();
    ui.addTagButton->setEnabled( true );
    ui.nextButton->setEnabled( true );
    ui.previousButton->setEnabled( true );
  }
}
```

تا به اینجا، توابع ما از کلاس ImageCollection کارهای مختلفی می‌خواستند، حال زمان آن رسیده که این کلاس را نیز تعریف و پیاده نماییم:

کلاس پایگاه داده

کلاس ImageCollection که شما را یک قدم به دیتابیس نزدیکتر می‌کند، مسئول تمامی ارتباطات با پایگاه داده ها است. هیچ نیازی نیست تا کلاسهای دیگر برنامه بدانند کلاس ImageCollection از یک پایگاه داده برای ذخیره اطلاعات استفاده میکند.

تعریف کلاس در ادامه می‌آید.

```
class ImageCollection
{
public:
      ImageCollection();
      QImage getImage( int id );
      QList<int> getIds( QStringList tags );
      QStringList getTags();
      void addTag( int id, QString tag );
      void addImage( QImage image, QStringList tags );
private:
      void populateDatabase();
};
```

توجه داشته باشید که اسم بعضی از توابع بصورت getXxx آمده است، که با روش کیوت برای نوشتن تابع getter فرق دارد، پس این نامگذاری به کلاسها نشان می‌دهد که این تابع یک تابع getter نیست، و این اطلاعات را از جایی(پایگاه داده/فایل/شبکه/الخ) کسب می‌کند، که این پروسه میتواند زمانبر باشد.

در این برنامه سعی شده هر تابع یک عمل محدود و مشخص را انجام دهد، که خواننده بتواند از روی نام تابع مسئولیت آنرا درک کند.

سازنده‌ی کلاس:

```
ImageCollection::ImageCollection()
{
   QSqlDatabase db = QSqlDatabase::addDatabase( "QSQLITE" );
   db.setDatabaseName( ":memory:" );
   if( !db.open() )
      qFatal( "Failed to open database" );
   populateDatabase();
}
```

سازنده کلاس یک دیتابیس را می‌سازد و آنرا باز می‌کند و بعد از آن دیگر توابع برنامه از دیتابیس پیشفرض استفاده می‌کنند، پس نیازی به نگهداری یک متغیر برای آن نیست. در اینجا دیتابیس را در حافظه نگهداری کردیم، یعنی بعد از پایان اجرای برنامه، اطلاعات آن از بین خواهد رفت، این عمل در زمان تمرین می‌تواند مفید باشد. البته که شما می‌توانید بجای :memory: آدرس یک فایل را به برنامه بدهید، و دیتابیس را نگهداری کنید.

سپس سازنده تابع populateDatabase را فراخوانی می‌کند، تا جداول پایگاه داده را بسازد. این تابع دو جدول می‌سازد، در اینجا از شرط IF NOT EXISTS استفاده کردیم، که برای حالتی که دیتابیس در حافظه است، زیاد ضروری نیست، اما اگر آنرا در فایل ذخیره کنیم، استفاده از این شرط لازم است.

```
void ImageCollection::populateDatabase()
{
  QSqlQuery qry;
  qry.prepare( "CREATE TABLE IF NOT EXISTS images "
               "(id INTEGER PRIMARY KEY, data BLOB)" );
  if( !qry.exec() )
     qFatal( "Failed to create table images" );
  qry.prepare("CREATE TABLE IF NOT EXISTS tags (id INTEGER, tag VARCHAR(30))");
  if( !qry.exec() )
     qFatal( "Failed to create table tags" );
}
```

کار کردن با تگ عکسها

یکی از مسئولیتهای ImageCollection نگهداری اینکه چه تگهایی داریم، و کدام تگ مربوط به کدام عکس است، می‌باشد.

سورس کد این تابع در ادامه می‌آید، چون ما سعی می‌کنیم اصولی پیش برویم، ابتدا کوئری را آماده می‌کنیم، سپس اجرا و در نهایت اطلاعات آنرا استفاده می‌کنیم. همینطور اینجا در کوئری از DISTINCT استفاده کردیم، که چک میکند که خروجی ما تگ تکراری نداشته باشد،(از جایی که ممکن است چندین عکس تگ تکراری داشته باشند.)

```
QStringList ImageCollection::getTags()
{
  QSqlQuery qry;
  qry.prepare( "SELECT DISTINCT tag FROM tags" );
  if( !qry.exec() )
     qFatal( "Failed to get tags" );
  QStringList result;
  while( qry.next() )
     result << qry.value(0).toString();
  return result;
}
```

تابع دیگری که برای مدیریت تگها استفاده می‌شود تابع addTag است، که یک تگ را به تگهای یک عکس می‌افزاید. که عکسی که این تگ به آن مربوط است، با idی آن مشخص می‌گردد. هیچ محدودیتی در مورد افزودن تگهای تکراری به عکس یکسان و غیریکسان وجود ندارد، بعلاوه اینکه تابع getTags چک می‌کند که تگ تکراری نگیرد!

```
void ImageCollection::addTag( int id, QString tag )
{
   QSqlQuery qry;
   qry.prepare( "INSERT INTO tags (id, tag) VALUES (:id, :tag)" );
   qry.bindValue( ":id", id );
   qry.bindValue( ":tag", tag );
   if( !qry.exec() )
      qFatal( "Failed to add tag" );
}
```

عکسها

تابع getIds به عکسها از زاویه‌ی تگها نگاه می‌کند، یعنی یک QStringList از تگهای مورد نظر می‌گیرد، و لیستی از idی عکسهایی که آن تگها را دارند، بر می‌گرداند. اگر هیچ تگی به این تابع داده نشود، یعنی لیست خالی باشد، لیستی از کلیه عکسهای پایگاه داده بر می‌گرداند. دلیل اینکه در سورس دو کوئری متفاوت را آماده کرده ایم این است.

در کوئری از IN استفاده کرده ایم، که ممکن است برای بعضی جدید باشد، نوشتن

```
x IN (1,2, 3)
```

مثل نوشتن:

```
x=1 OR x=2 or x=3
```

است.

از جایی که واسط برنامه مطمئن می‌شود که تگها فقط حروف a تا z هستند، شما می‌توانید با خیال راحت آنها را به هم چسبانده در کوئری ها استفاده نمایید.

نکته: از وارد کردن رشته بطور دستی در یک کوئری پرهیز کنید، همیشه از bindValue و توابع مشابه استفاده کنید.

```
QList<int> ImageCollection::getIds( QStringList tags )
{
   QSqlQuery qry;
   if( tags.count() == 0 )
      qry.prepare( "SELECT images.id FROM images" );
   else
      qry.prepare( "SELECT id FROM tags WHERE tag IN ('" +
                   tags.join("','") + "') GROUP BY id" );
   if( !qry.exec() )
      qFatal( "Failed to get IDs" );
   QList<int> result;
   while( qry.next() )
      result << qry.value(0).toInt();
   return result;
}
```

ذخیره کردن عکس در پایگاه داده

ذخیره کردن عکس در پایگاه داده یک عمل سر راست و مستقیم نیست، چون هیچ نوع داده ای برای ذخیره کردن عکس در SQLite تعریف نشده است، در عوض ما به نوع داده ای BLOB که برای نگهداری اطلاعات باینری بزرگ است، اعتماد می‌کنیم.

پروسه ذخیره کردن یک QImage در یک blob در دیتابیس، می‌تواند به سه قسمت تقسیم گردد. ابتدا شما یک بافر در حافظه ایجاد می‌کنید، و عکس را در آن ذخیره می‌کنید. سپس بافر به یک QByteArray تبدیل می‌گردد، که به یکی از متغیرهای کوئری مربوط می‌گردد.

همه‌ی اینها در تابع addImage انجام می‌پذیرد.

```
void ImageCollection::addImage( QImage image, QStringList tags )
{
  QBuffer buffer;
  QImageWriter writer(&buffer, "PNG");
  writer.write(image);
  QSqlQuery qry;
  int id;
  qry.prepare( "SELECT COUNT(*) FROM images" );
  qry.exec();
  qry.next();
  id = qry.value(0).toInt() + 1;
  qry.prepare( "INSERT INTO images (id, data) VALUES (:id, :data)" );
  qry.bindValue( ":id", id );
  qry.bindValue( ":data", buffer.data() );
  qry.exec();
  foreach( QString tag, tags )
     addTag( id, tag );
}
```

همانطور که ملاحظه می‌کنید، قبل از افزودن عکس، تعداد عکسها را از دیتابیس می‌گیریم، اینکار برای محاسبه‌ی id برای عکس جدید است. اگر اجازه‌ی حذف کردن عکسها را به کاربر بدهیم، این سیستم کار نخواهد کرد، و شما باید از AUTOINCREMENT در زمان ساختن جدول استفاده کنید، تا خود SQLite افزودن شماره id را مدیریت کند.

در اینجا کوئری INSERT کاملا گویا است.

پروسه‌ی گرفتن عکس از پایگاه داده، و نگهداری آن در یک QImage در همان کلاس و در تابع getImage انجام می‌گردد. این عملیات ساده تر از ذخیره کردن است، و قاعدتا کد آن دیگر باید برای شما مفهوم باشد.

```
QImage ImageCollection::getImage( int id )
{
   QSqlQuery qry;
   qry.prepare( "SELECT data FROM images WHERE id = :id" );
   qry.bindValue( ":id", id );
   if( !qry.exec() )
      qFatal( "Failed to get image" );
   if( !qry.next() )
      qFatal( "Failed to get image id" );
   QByteArray array = qry.value(0).toByteArray();
   QBuffer buffer(&array);
   buffer.open( QIODevice::ReadOnly );
   QImageReader reader(&buffer, "PNG");
   QImage image = reader.read();
   return image;
}
```

گذاشتن اجزاء کنار هم

کلاس ImageDialog یک نسخه از کلاس ImageCollection برای رتق و فتق امور خود دارد، پس تنها کاری که برای اجرای برنامه باقی مانده این است که تابع main یک Qapplication و یک ImageDialog بسازد، سپس پنجره‌ی برنامه که همانا ImageDialog است را نشان بدهد و برنامه را وارد چرخه رخداد(event loop) نمایید:

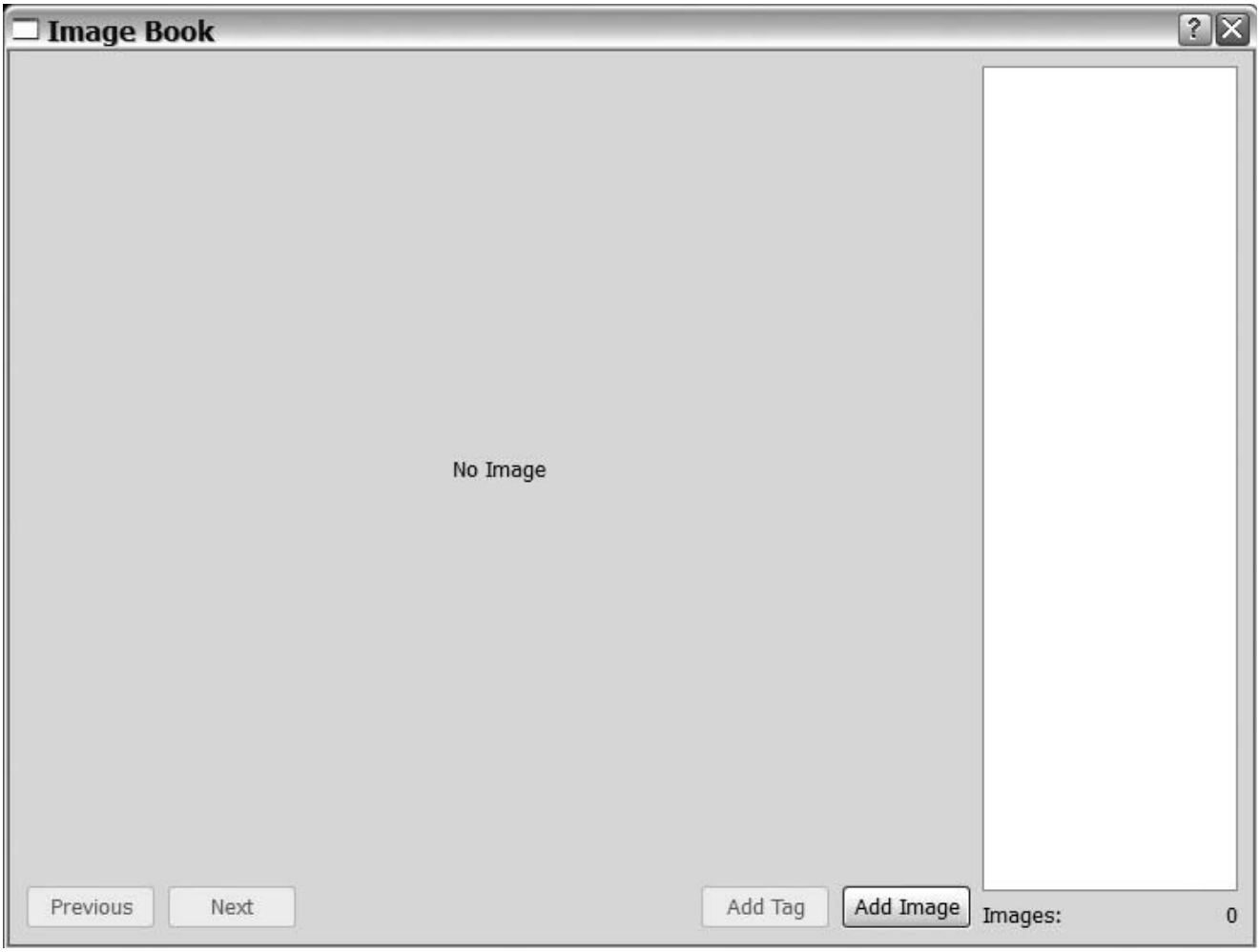
```
int main( int argc, char **argv )
{
   QApplication app( argc, argv );
   ImageDialog dlg;
   dlg.show();
   return app.exec();
}
```

فایل پروژه با اجرای qmake –project ایجاد می‌گردد. پس از آن شما باید خط زیر را به این فایل بیافزایید:

```
QT += sql
```

اینگونه به qmake می‌گوییم که در این برنامه از ماژول QtSql هم استفاده کرده ایم، پس لازم است که کتابخانه‌ی مربوطه را نیز در زمان لینک به برنامه لینک نماید.

سپس با اجرای qmake و make برنامه آماده‌ی استفاده است.



در Qt کلاسها و ابزارهای مفید دیگری نیز برای کار با پایگاه داده ها آماده شده اند، که با مراجعه به مستندات آن می‌توانید فواید و نحوه استفاده از آنها را بیابید.

فصل هفتم: بررسی چند تکنولوژی دیگر کیوت

کیوت ابزارها و چارچوبهای زیادی در دسترس برنامه‌نویس قرار می‌دهد، که توضیح کامل و معرفی همه‌ی آن‌ها از بحث یک کتاب کامل هم فراتر می‌رود. در این فصل بیشتر به معرفی دیگر تکنولوژی‌های کیوت می‌پردازیم تا آموزش استفاده از آنها، تا شما از وجود همچین ابزارهایی مطلع گردید و در زمان لازم مستندات آن‌ها را از طریق مستندات مرجع کیوت خوانده و از آن‌ها استفاده کنید.

ماژول QtNetwork

این ماژول کلاسهایی برای نوشتن کلاینت و سرور TCP/IP در اختیار شما قرار می‌دهد. کلاسهایی چون QFtp مخصوص نوشتن برنامه‌هایی با هدف خاص، کلاسهایی برای کار با لایه‌های پایین‌تر مثل ، QTcpSocket QTcpServer و QUdpSocket که مفاهیم سطح پایین شبکه را می‌فهمند. و کلاسهایی برای استفاده از مفاهیم سطح بالا، مثل QNetworkReply ، QNetworkRequest و QNetworkAccessManager

عملیات سطح بالای شبکه برای Http و Ftp

APIی دسترسی به شبکه یک کلکسیون از کلاسها برای انجام عملیات عمومی شبکه است. API یک لایه‌ی مفهومی روی لایه‌ی عملیاتی و پروتکلهای شبکه تشکیل داده است(برای مثال گرفتن و فرستادن اطلاعات از طریق HTTP). و تنها کلاسها، توابع و سیگنالهایی برای عملیات عمومی و عملیات سطح بالا را در اختیار قرار می‌دهد.

درخواستهای شبکه با کلاس QNetworkRequest نشان داده می‌شوند. که بعنوان یک نگهدارنده‌ی عمومی برای اطلاعاتی که به یک درخواست مربوط می‌شوند(مثل اطلاعات header و رمزنگاری مورد استفاده) عمل می‌کند. پروتکل مورد استفاده از طریق URLای که به درخواست اختصاص داده شده‌است تشخیص داده می‌شود، در حال حاضر تنها پروتکلهای HTTP و FTP و فایل(File) (برای دسترسی به فایلهای سیستم) در دسترس می‌باشند.

هماهنگی عملیات شبکه با استفاده از کلاس QNetworkAccessManager انجام می‌گردد. زمانی که یک درخواست ایجاد می‌گردد، یک نسخه از این کلاس در اختیار برنامه‌نویس قرار می‌گیرد تا اطلاعات مربوط به اجرای آنرا در اختیار قرار دهد. این کلاس همچنین استفاده و ذخیره کوکی، پروکسی و تصدیق اطلاعاتی کاربر را نیز مدیریت می‌کند.

پاسخ درخواست‌های شبکه در قالب کلاس QNetworkReply شناخته می‌شوند. زمانی که یک درخواست به نتیجه می‌رسد، QNetworkAccessManager یک نمونه از این کلاس را ساخته و در اختیار برنامه‌نویس قرار می‌دهد.

ماژول QtXml

این ماژول مخصوص انجام عملیات خواندن و نوشتن متون به فرمت XML می‌باشد. این ماژول عملیات مربوط به نگهداری اطلاعات در فرمت XML را به طور کامل پشتیبانی می‌کند.

رخدادها

علاوه بر مفهوم سیگنال و اسلات، در کیوت مفهوم رخداد (event) نیز وجود دارد، که اشیائی هستند که از کلاس QEvent مشتق شده، و هرکدام یک اتفاق خارجی یا داخلی را نشان می‌دهند. رخدادها بیشتر در مورد ویدجتها مورد استفاده قرار می‌گیرند، مثلاً رخداد ورود نشانگر موس به ویدجت، یا خروج آن.

برای استفاده از رخدادهای یک ویدجت، باید یک subclass (کلاسی که وارث خصوصیاتی از یک کلاس دیگر می‌شود) از آن بسازید، و درکلاس شخصی شده از رخدادها استفاده کنید، بعنوان مثال کد زیر رخداد کلیک شدن موس روی یک check box را مدیریت می‌کند:

```
void MyCheckBox::mousePressEvent(QMouseEvent *event)
 {
     if (event->button() == Qt::LeftButton) {
         // handle left mouse button here
     } else {
         // pass on other buttons to base class
         QCheckBox::mousePressEvent(event);
     }
 }
```

شما می‌توانید یک رخداد مخصوص برنامه‌ی خود بسازید، و همانند کیوت آنرا در زمان مورد نظر ارسال کنید، با استفاده از تابع QCoreApplication::sendEvent() و یا QCoreApplication::postEvent() فعال نمایید.

برنامه‌نویسی چند نخی (Multi-threading Programming)

کیوت امکان نوشتن برنامه‌های چند نخی بدون وابستگی به سیستم‌عامل را فراهم کرده است.

اگر با برنامه‌نویسی چند نخی آشنا باشید، کیوت همان مفاهیم را در کلاسهایی با همان نام‌ها آماده کرده است، بعنوان مثال کلاس QMutex که برای پیاده‌سازی مفهوم mutexها که جهت ارتباط بین نخ‌های برنامه مورد استفاده قرار می‌گیرد، و یا کلاس QSemaphore و... آماده شده است. کلاس QThread نیز دیگر کلاس مهم این مجموعه می‌باشد، که مفهوم یک نخ را نشان می‌دهد و ابزارهای مورد نظر برای مدیریت یک نخ برنامه را آماده کرده است، مثل تابع start که همان تابع run که معادل تابع main در یک برنامه است را فراخوانی می‌کند.

QtWebKit

QtWebKit یک موتور مرورگر وب آماده است، که می‌توان در برنامه از آن استفاده کرد. و استفاده‌های زیادی می‌توان از این ابزار کرد.

بعنوان یک مثال ساده، کد زیر را ببینید:

```
   QWebView *view = new QWebView(parent);
   view->load(QUrl("http://qt.nokia.com/"));
   view->show();
```

ذخیره و بازیابی تنظیمات کاربر

کلاس QSettings به عنوان ابزاری برای نگهداری و مدیریت تنظیمات برنامه، در اختیار برنامه‌نویس قرار دارد.
مثالهای زیر را ببینید:

```
QSettings settings("MySoft", "Star Runner");
 QColor color = settings.value("DataPump/bgcolor").value<QColor>();
```

```
QSettings settings("MySoft", "Star Runner");
 QColor color = palette().background().color();
 settings.setValue("DataPump/bgcolor", color);
```

یا کد زیر جهت ذخیره و بازیابی اندازه و محل یک برنامه(روی صفحه‌ی نمایش) مورد استفاده قرار می‌گیرد:

```
void MainWindow::writeSettings()
 {
     QSettings settings("Moose Soft", "Clipper");

     settings.beginGroup("MainWindow");
     settings.setValue("size", size());
     settings.setValue("pos", pos());
     settings.endGroup();
 }

 void MainWindow::readSettings()
 {
     QSettings settings("Moose Soft", "Clipper");

     settings.beginGroup("MainWindow");
     resize(settings.value("size", QSize(400, 400)).toSize());
     move(settings.value("pos", QPoint(200, 200)).toPoint());
     settings.endGroup();
 }
```

Phonon

با استفاده از چارچوب Phonon شما می‌توانید به راحتی برنامه‌هایی برای خواندن و اجرا کردن فایل‌های صوتی و تصویری بنویسید، کد زیر را ببینید:

```
   Phonon::VideoPlayer *player =
         new Phonon::VideoPlayer(Phonon::VideoCategory, parentWidget);
   player->play(url);
```

کاملاً مشخص است که یک player تصویری ساخته شده و فایلی را اجرا نموده است.
البته جهت استفاده از این ماژول باید خط زیر را به فایل پروژه بیافزایید:

```
   QT += phonon
```

DBus و برقرار ارتباط بین برنامه‌ها

اگر با مفهوم DBus که یکی از ابزارهای ارتباط بین برنامه‌ها در لینوکس است، آشنا باشید، دانستن اینکه ماژولی بنام QtDBus جهت استفاده از آن در کیوت موجود است برایتان جالب خواهد بود. که تمامی مفاهیم مربوطه از سیگنالها، پیغام‌ها، درخواست‌ها و جوابها را پشتیبانی می‌کند.

برای استفاده از این ماژول نیز افزودن dbus به متغیر QT در فایل پروژه الزامی است.

```
QT += dbus
```

علاوه بر موارد مطرح شده، تکنولوژی‌ها، چارچوب‌ها و ماژول‌های دیگری نیز جهت ساده و سریع سازی کار برنامه‌نویس آماده شده‌اند که در http://qt.nokia.com/doc/latest/best-practices.html و http://doc.qt.nokia.com/latest/frameworks-technologies.html می‌توانید آن‌ها را بیابید.

فصل هشتم: استفاده از مستندات مرجع Qt

مستندات مرجع کیوت، یک ابزار و منبع مهم و حیاتی برای هر برنامه‌نویس کیوت می‌باشد. در این کتابچه شما با بعضی از کلاسها و توابع کیوت آشنا می‌شوید، اما کلاسها و توابع زیاد دیگری هستند، که در آینده به آن‌ها نیاز خواهید داشت، بهترین منبع و مرجع برای اینکار مستنداتی است که همراه با کیوت منتشر می‌گردند. و با هر انتشار به روز می‌گردند. برای هر کلاس و تابع‌ای که در کتابخانه‌ی کیوت می‌بینید حداقل یک خط و گاهی چندین مثال و آموزش در این مرجع وجود دارد.

پس برای شما هم واجب است هرچه سریعتر استفاده از این منبع را یاد بگیرید.

در اکثر توزیع‌های لینوکس، این مستندات بصورت یک بسته جدا از کتابخانه بسته‌بندی و آماده‌ی نصب می‌گردد، بعلت حجم بالای آن و اینکه کاربران برنامه‌هایی که با کیوت نوشته‌شده‌اند نیازی به این مستندات ندارند. اصولاً تحت نام qt-doc و نامهایی شبیه به آن منتشر می‌گردد.

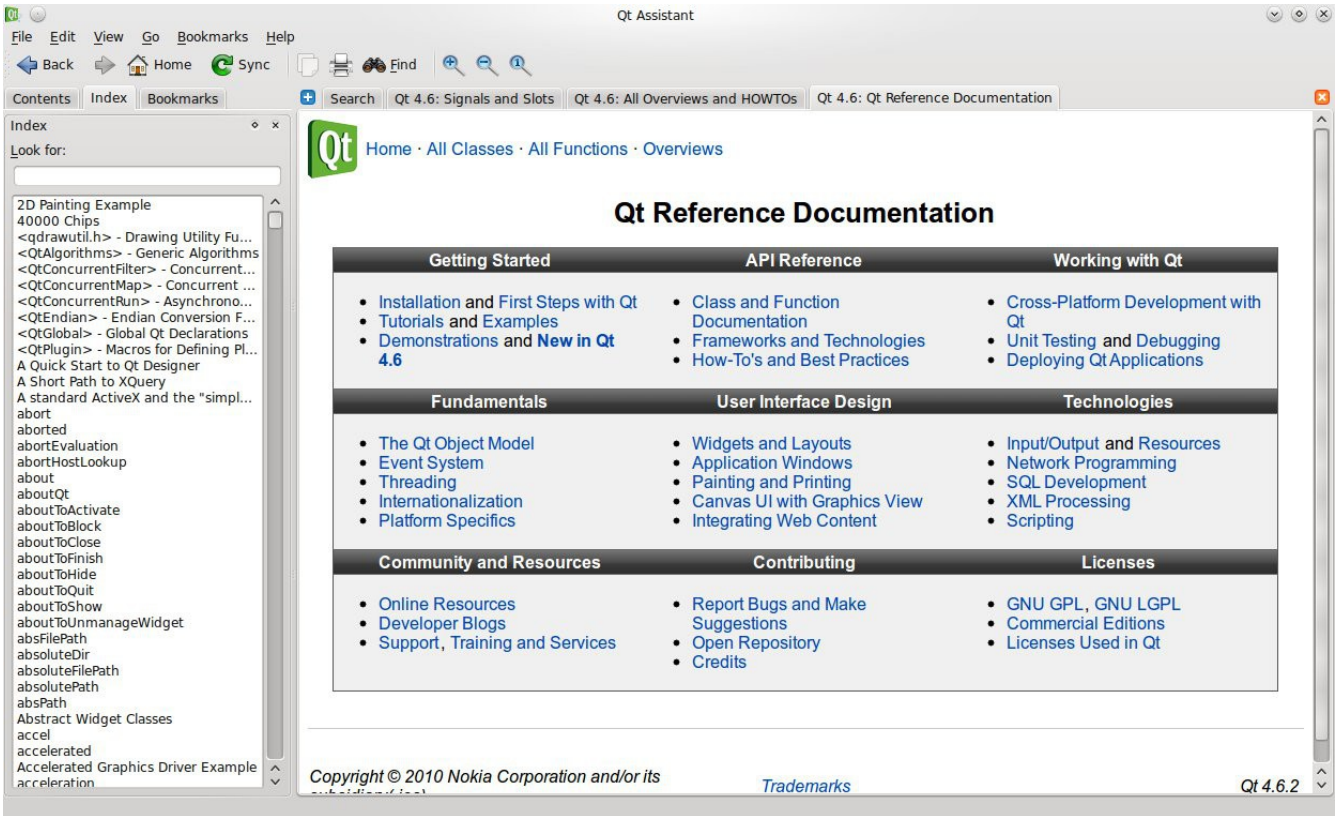
در ویندوز، همراه Qt SDK عرضه می‌گردد.

این مستندات هم به فرمت html که قابل استفاده بوسیله‌ی یک مرورگر وب است و هم به فرمت خاص کیوت که برنامه‌ی Qt Assistant که جزئی از SDKی توسعه‌ی برنامه بوسیله‌ی Qt است، عرضه می‌گردند. همیشه می‌توانید به آخرین نسخه‌های مستندات در آدرس http://qt.nokia.com/doc مراجعه کنید.

اما ابزار Assistant بعلت داشتن امکانات ایندکس‌گذاری و جستجو در مستندات، بسیار مفیدتر خواهد بود.

برای اجرای آن کافی است دستور assistant را در لینوکس و یا برنامه را از منوی استارت ویندوز اجرا کنید.

پنجره‌ی این برنامه:



این مستندات علاوه بر اطلاعات مرجع در مورد همه‌ی کلاسها و توابع کیوت، تعداد قابل توجهی آموزش در مورد کارهای مختلفی که با کیوت می‌توان انجام داد، و مثالی برای هر موضوع و اطلاعات مفید دیگری در مورد این چارچوب، دارا است.

هرگاه نیاز به خواندن مستندات یک کلاس یا تابع دارید، کافیست نام آنرا در قسمت index وارد کنید. و به سادگی صفحه‌ی مستندات آنرا ببینید.

می‌توانید برای شروع سری به مستندات کلاسها و توابعی که تاکنون استفاده کرده‌اید بزنید، تا اطلاعات کاملتری در مورد آن‌ها بیابید.

همینطور شما از داخل ابزارهای مدرنی مثل KDevelop و Qt Creator به سادگی به این مستندات دسترسی دارید. بعنوان مثال داخل Creator وقتی اطلاعات در مورد یک کلاس می‌خواهید کافیست اشاره‌گر ویرایشگر را روی آن نگهداشته کلید F1 را فشار دهید.

منابع

1. Foundations of Qt Development, Johan Thelin, 2007
2. C++ GUI Programming with Qt4, Jasmin Blanchette, Mark Summerfield, 2006
3. Qt Documentation References, Qt Software, Nokia Inc. 2010
4. Qt Tutorials and Examples, Qt Software, Nokia Inc. 2010