آشنایی با Qt

تهیه و تنظیم : عباس نادری
abiusx@etebaran.com
پاییز ۱۳۸۸
نسخه اول

Qt یک چهارچوب توسعه نرم‌افزار میان‌سکویی (Cross Platform) که معمولا برای توسعه نرم‌افزارهای واسط گرافیکی (GUI) استفاده می‌شود. Qt بسیار کاملتر از مجموعه ابزارهای دیگر مانند GTK است که تنها مجموعه‌ای از اشیای گرافیکی قابل استفاده بر روی پنجره‌ها هستند. Qt از زبان سی‌پلاس‌پلاس استفاده می‌کند ولی تبدیل آن بر روی اکثر زبان‌های برنامه‌نویسی تولید شده است. همچنین اکثر ویژگیهای کیوت از Preprocessor موجود در زبان سی استفاده می‌کنند و برخی از ویژگیهای خاص آن (مانند پنجره‌ساز و چندزبانی) از یک پیش پردازنده که مخصوص خود کیوت است و کد آنرا به کد سی تبدیل می‌کند، استفاده می‌کنند.

مانند اکثر چهارچوب‌های برنامه‌نویسی واسط گرافیکی (از جمله Microsoft Visual Studio, Apple Xcode, ...) برنامه کیوت از یک حلقه رخداد اصلی (Main Event Loop) استفاده می‌کند. به دلیل متن‌باز بودن کیوت و رشد آن در طی سالهای اخیر، این حلقه بسیار بهینه پیاده‌سازی شده و از مشابه خود در سیستم‌های مذکور بسیار بهتر عمل می‌کند. مانند اکثر محیط‌های برنامه‌سازی گرافیکی، کیوت نیز از برنامه‌نویسی رخداد محور (Event-Driven Programming) پشتیبانی می‌کنند ولی الگوی مشاهدگر (Observer Pattern) که یکی از الگوهای بسیار عالی و موفق در مهندسی نرم‌افزار است امکانات و افق‌های بسیاری را به روی برنامه‌نویس می‌گشاید.

جهت کار با کیوت باید Qt SDK را که حجم قابل توجهی نیز دارد (۳۰۰ الی ۷۰۰ مگابایت) برای سکوی خود دانلود کنید (سکو = سخت افزار +سیستم عامل) www.qtsoftware.com

پروژه در Qt

به مثابه تمام محیط‌های برنامه‌سازی حرفه‌ای، کیوت نیز برنامه‌ها را در قالب پروژه نگهداری می‌کند. انواع متنوعی از پروژه‌ها در کیوت وجود دارند ولی مقصود ما معمولا پروژه GUI Application است. جهت ایجاد یک پروژه جدید از درون Qt Creator گزینه New Project و Qt GUI Application را انتخاب نمایید. در مرحله بعدی نامی و جایی برای ذخیره شدن فایلهای پروژه تعیین کنید. به نام هر پروژه یک پوشه ساخته می‌شود و فایلهای آن درون پوشه قرار می‌گیرند. دقت کنید که محل ساده‌ای برای جای پروژه انتخاب کنید (زیاد تودرتو نباشد)

در مرحله بعدی جادوگر، ماژولهایی که می‌خواهید از چهارچوب کیوت به برنامه خود بیافزایید انتخاب می‌کنید. دقت کنید که بعدا می‌توانید تغییرات لازم را اعمال کنید بنابراین تنظیمات پیش فرض (یعنی QtCore جهت کارهای حیاتی کیوت و QtGui جهت برنامه گرافیکی و پنجره‌ها) مناسب است.

قسمت بعدی سه نوع پنجره برای شروع به شما پیشنهاد می‌دهد، QMainWindow, QDialog ,. QWidget ویجت به هر شیء که بر روی یک پنجره قرار می‌گیرد اتلاق می‌شود و یک اصطلاح کلیست. (هر شیء که ظاهری گرافیکی دارد) دیالوگ به یک پنجره ساده گفته می‌شود. MainWindow یک دیالوگ است که نوار ابزار، منو و نوار وضعیت در آن گنجانده شده است، بنابراین جهت جلوگیری از پیچیدگی دیالوگ توصیه می‌شود.

در مرحله بعدی پروژه ساخته می‌شود. در پروژه چندین فایل اولیه قرار خواهد داشت، به عنوان مثال اگر نام پروژه project باشد فایلهای زیر در پروژه موجودند:

project.pro فایل معرف پروژه است که برای کامپایل کردن از اطلاعات آن استفاده می‌شود. اینکه چه ماژول‌هایی و چه تنظیماتی استفاده شده در این فایل به صورت متن ساده وجود دارد. آنرا باز کنید واضح است.

main.cpp تابع اصلی برنامه و نقطه شروع آن در این فایل است. معمولا در این فایل یک متغیر از نوع QApplication ساخته شده، یک دیالوگ نیز ساخته می‌شود (فایل دیالوگ include می‌شود) ، دیالوگ نمایش داده می‌شود و با استفاده از متغیر از نوع برنامه، حلقه رخداد آغاز می‌شود.

dialog.h , dialog.cpp فایلهای هدر و سورس کلاس دیالوگتان، که می‌توانید کدهای مربوط به دیالوگ را در آنها اضافه کنید. به عنوان مثال کدهای رخدادهایی که می‌خواهید مدیریت کنید و یا سیگنال‌ها و شیارهای پیاده‌سازی الگوی مشاهدگر.

dialog.ui فایلهای ui فایلهای Qt Designer هستند، برنامه‌ای که با استفاده از آن می‌توانید به صورت تصویری یک دیالوگ بسازید و خودش دیالوگ ساخته شده شما را به صورت XML ذخیره کرده، هنگام کامپایل به کد سی‌پلاس‌پلاس ترجمه می‌کند.

main.cpp

کد فایل اصلی پروژه به صورت زیر است:

```
#include <QtGui/QApplication>
#include "dialog.h"

int main(int argc, char *argv[])
{
```

```
    QApplication a(argc, argv);
    Dialog w;
    w.show();
    return a.exec();
}
```

خط اول جهت ضمیمه کردن کدهای مربوط به QtCore است که یک برنامه GUI را مدیریت می‌کند. هنگامی که این include انجام شود متغیرهایی از نوع QApplication قابل تعریف می‌شوند.
خط بعدی ضمیمه کردن کد مربوط به دیالوگ شماست. از آنجایی که پس از شروع برنامه دیالوگ باید نمایش داده شود، لازم است تا متغیری از نوع دیالوگ نیز تعریف کنیم و آنرا نمایش دهیم.
در مابقی خطوط برنامه یک تابع اصلی استاندارد سی‌پلاس‌پلاس را می‌بینیم که در آن یک متغیر از نوع QApplication گرفته شده (و پارامترهای ورودی برنامه CLA به آن ارسال شده) و سپس یک دیالوگ از نوعی که تعریف کرده‌ایم به نام w تعریف شده، دیالوگ نمایش داده شده و سپس حلقه رخداد با تابع a.exec فراخوانی می‌شود. هنگامی که تابع حلقه رخداد به انتهای اجرای خود برسد، برنامه خاتمه می‌یابد.
در نظر داشته باشید که درخت پروژه از این فایل شروع می‌شود و فایل dialog.h را در برمی‌گیرد.

Project.pro

```
#-------------------------------------------------
#
# Project created by QtCreator ۱۷-۱۲-۲۰۰۹T۲۰:۵۳:۰۷
#
#-------------------------------------------------

TARGET = Project
TEMPLATE = app


SOURCES += main.cpp\
        dialog.cpp

HEADERS  += dialog.h

FORMS    += dialog.ui
```

فایل پروژه حامل چند خط تنظیمات است. هر فایلی که به پروژه بیافزایید، Qt Creator به صورت خودکار آنرا به این تنظیمات می‌افزاید. همچنین هر ماژولی که خواستید بیافزایید یا کم کنید، با استفاده از عبارت‌های زیر ممکن است:

QT += sql

QT -= gui

توجه داشته باشید که محتوای این فایل تنظیماتیست که توسط پیش کامپایلر کیوت استفاده می‌شود و برنامه سی نیست.

dialog.ui

هر دفعه که برنامه را کامپایل می‌کنید (فرآیند Build) از هر فایل ui یک فایل h ساخته می‌شود که کدهای مربوط به دیالوگی که به صورت تصویری ساخته‌اید درون آن قرار می‌گیرند. اگر گزینه hide generated files را بردارید، این فایل نیز به شما نمایش داده می‌شود. قانونی که کیوت استفاده می‌کند آنست که به ازای فایل ui ای به نام somefile.ui یک فایل h با نام ui_somefile.h می‌سازد که شما می‌توانید آنرا در مابقی کد خود include کرده و استفاده کنید. معمولا کل این فرآیند و حتی استفاده از این فایل به صورت خودکار توسط کیوت برای شما انجام می‌شود.

dialog.h

```
#ifndef DIALOG_H
#define DIALOG_H

#include <QDialog>

namespace Ui {
    class Dialog;
}

class Dialog : public QDialog {
    Q_OBJECT
public:
    Dialog(QWidget *parent = 0);
    ~Dialog();
```

```
protected:
    void changeEvent(QEvent *e);

private:
    Ui::Dialog *ui;
};

#endif // DIALOG_H
```

فایل تعریف دیالوگ شما با نام dialog.h ساخته شده است. هر موقعی که دیالوگی به برنامه اضافه کنید، باید نامی برای آن انتخاب کنید. هر نامی که انتخاب کنید سه فایل با آن نام با فرمت‌های h , cpp , ui ساخته خواهد شد.

در ابتدا و انتهای این فایل ساختارهای جلوگیری از چندبار ضمیمه شدن فایل با استفاده از # را می‌بینیم که اتوماتیک توسط کیوت افزوده شده است. در قسمت بعدی کتابخانه QDialog کیوت که مسئول تعریف کردن موارد مربوط به یک دیالوگ است ضمیمه شده است.

قسمت سوم که شامل سه خط کد است، پروتوتایپ استفاده از کد تولید شده از روی فایل ui را در بر دارد. دقت کنید که Dialog در فضای نام Ui همان دیالوگیست که در فایل dialog.ui تعریف می‌شود و Dialog در فضای نام اصلی برنامه، چیزیست که در ادامه این فایل تعریف شده است.

در ادامه دیالوگ مربوط به ما تعریف شده است و از QDialog که دیالوگ پیش فرض و استاندارد Qt است ارث برده است (یعنی تمام ویژگیهای آنرا نیز در بر دارد) . این فرآیند به ما کمک می‌کند تا کد زیادی برای دیالوگ خودمان ننویسیم، بلکه تنها قسمت‌هایی را که می‌خواهیم با یک دیالوگ استاندارد تفاوت داشته باشد بنویسیم (مابقی به واسطه وراثت از دیالوگ استاندارد در دیالوگ ما کپی می‌شوند). تمام کد دیالوگ باید در داخل این قسمت نوشته شود. (البته به دلیل بالا بردن خوانایی برنامه، پروتوتایپ توابع و متغیرهای دیالوگمان در فایل h تعریف می‌شوند و بدنه توابع در فایل cpp نوشته می‌شود. به همین دلیل دو فایل cpp , h برای هر دیالوگ تولید می‌شوند. شما می‌توانید این فایلها را ادغام کنید تا تنها یک فایل داشته باشید ولی برای حفظ خوانایی هماهنگی با این روش توصیه می‌شود).

در تمام کلاس‌هایی که از یک کلاس کیوت ارث می‌برند و استفاده می‌کنند، باید عبارت Q_OBJECT به عنوان اولین خط وجود داشته باشد. این عبارت توسط پیش پردازنده کیوت استفاده می‌شود. در صورتی که این عبارت فراموش شود خطاهای عجیب و نامفهومی دریافت خواهید کرد. خوشبختانه کیوت به صورت پیشفرض این کدها را تولید می‌کند و تنها کفایت می‌کند ما حذفشان نکنیم!

صرفنظر از بخشهای public, protected , private تمام توابع و متغیرهایی که در داخل کلاس تعریف می‌شوند در داخل کلاس قابل استفاده هستند. در واقع دنیای اکثر برنامه‌ها داخل یک دیالوگ است و تعاملات محدودی بین دیالوگ‌ها وجود دارد (مگر در موارد خاص)

توابع Dialog و Dialog~ به ترتیب هنگام گرفتن متغیر از نوع دیالوگ و پاک شدن آن متغیر از حافظه فراخوانی خودکار خواهند شد. تابع changeEvent برای چندزبانی کردن برنامه به‌کار می‌رود و به آن کاری نداریم. متغیر ui از نوع Ui::Dialog ساخته شده است، یعنی دیالوگی که با Qt Designer به صورت تصویری ساختیم. قابل توجه است که دیالوگ ما دیالوگی که به صورت تصویری ساخته‌ایم را به صورت یک متغیر در داخل خود دارد و بنابراین می‌تواند به تمام بخش‌های آن دسترسی داشته باشد. پروتوتایپ دیالوگ ما در همینجا خاتمه می‌یابد و مابقی کد در فایل dialog.cpp قرار دارد.

توجه داشته باشید که برای اینکه یک متغیر به دیالوگتان بیافزایید باید آنرا در این فایل در داخل قسمت مربوطه تعریف کنید. همچنین اگر بخواهید یک تابع به دیالوگ خود بیافزایید، باید پروتوتایپ آنرا در این فایل و بدنه آنرا در dialog.cpp بنویسید. (البته می‌توانید پروتوتایپ و بدنه را یکجا در فایل h بنویسید)

dialog.cpp

```
#include "dialog.h"
#include "ui_dialog.h"

Dialog::Dialog(QWidget *parent) :
    QDialog(parent),
    ui(new Ui::Dialog)
{
    ui->setupUi(this);
}

Dialog::~Dialog()
{
    delete ui;
}

void Dialog::changeEvent(QEvent *e)
{
    QDialog::changeEvent(e);
    switch (e->type()) {
    case QEvent::LanguageChange:
        ui->retranslateUi(this);
        break;
    default:
```

```
        break;
    }
}
```

کد اصلی توابع در این فایل نوشته می‌شوند. مابقی برنامه معمولا طراحیست و این فایل برنامه‌نویسی. دقت کنید که چون کد این فایل مربوط به یک کلاس و یک دیالوگ است قاعدتا باید در یک فایل نوشته شود که معمولا بسیار طولانی می‌شود و نمی‌توان از ساختارهای include به خوبی در این فایل استفاده نمود.

در ابتدای این فایل، فایل dialog.h که حاوی پروتوتایپ‌های این فایل است ضمیمه شده (دقت کنید که dialog.h ساختاری داشت که از چندبار انضمام آن جلوگیری می‌کرد بنابراین این انضمام بی‌خطر و برای تضمین وجود پروتوتایپ‌ها قبل از تعریف توابع است)

خط دوم فایل مربوط به Designer را ضمیمه می‌کند. اگر به خاطر داشته باشید یک متغیر از این نوع در داخل دیالوگ خود گرفته بودیم، ولی چون این نوع وجود نداشت باید فایل آنرا ضمیمه کنیم تا بتوانیم متغیری از نوع آن استفاده کنیم. از این پس بدنه توابع مربوط به دیالوگ امان را تعریف می‌کنیم. از آنجایی که نام دیالوگ ما Dialogبود (یادآوری class Dialog:public QDialog) باید قبل از نام توابع نام دیالوگ به انضمام دو دونقطه را بنویسیم تا مشخص شود که این تابع مربوط به آن کلاس است، به عنوان مثال Dialog::myFunction . دقت داشته باشید که به ازای هر پروتوتایپی که در dialog.h تعریف کردیم یک تابع در این فایل باید بنویسیم و به ازای هر بدنه تابعی که در این فایل می‌نویسیم باید دقیقا یک پروتوتایپ در dialog.h بنویسیم.

تابع Dialog که هنگام ساخته شدن دیالوگ اولین چیزیست که فراخوانی می‌شود، مسئولیت ساختن پنجره را دارد. در قسمت اول آن QDialog را که دیالوگ استاندارد است فراخوانی می‌کنیم تا دیالوگ استاندارد ساخته شود. در قسمت بعدی متغیر ui را که از نوع دیالوگ تصویریمان گرفته بودیم مقداردهی می‌کنیم. سپس تابع setupUi را از این متغیر فراخوانی می‌کنیم. پس از این مراحل پنجره ما به همراه تمام چیزهایی که به صورت تصویری ساخته بودیم ساخته و تنظیم می‌شوند (کدهای تنظیم به صورت خودکار توسط Designer درون setupUi قرار می‌گیرند)

در تابع تخریب یعنی Dialog~، متغیر ui یعنی پنجره تصویری خود را پاک می‌کنیم تا حافظه آن آزاد شود. اگر به بدنه تابع changeEvent دقت کنید مشخص است که برای تنظیم تغییر زمان طراحی شده است و برای ما کاربرد خاصی ندارد بنابراین آنرا بررسی نمی‌کنیم.

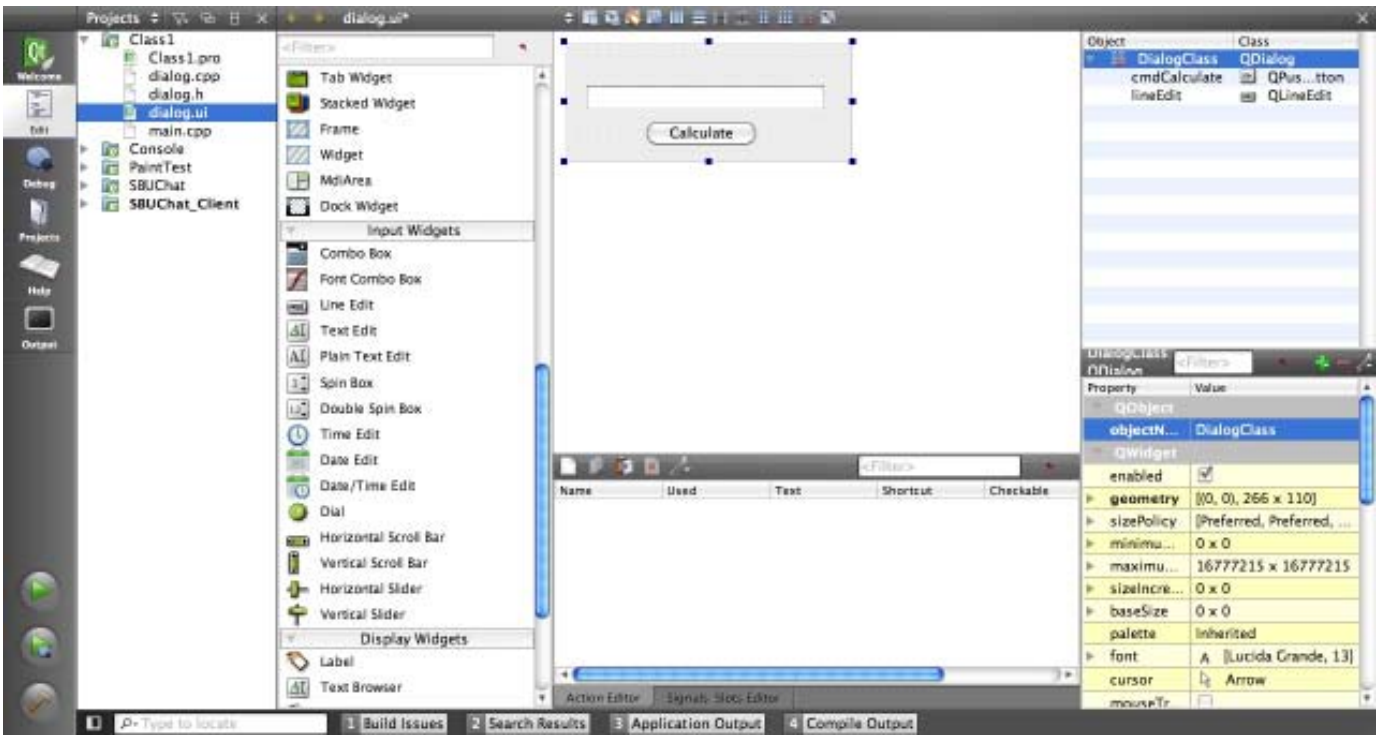
شروع کار

می‌خواهیم برنامه‌ای بنویسیم که یک عدد ورودی بگیرد و با کلیک بر روی یک کلید، رادیکال آنرا به صورت یک پیام نمایش دهد. برای این منظور ابتدا به سراغ فایل ui و Designer می‌رویم و موارد لازم را بر روی Ui Dialog اضافه می‌کنیم.

یک PushButton از لیست Widget ها در سمت چپ Designer (برای روی کار آمدن Designer بر روی فایل ui در لیست فایلهای پروژه دوکلیک کنید) بر روی دیالوگ می‌کشیم و قرار می‌دهیم. با دوکلیک بر روی آن نوشته روی آنرا به Calculate تغییر می‌دهیم. همچنین در لیست

پایین راست، که به Property Editor در همه محیط‌های دیداری برنامه‌سازی مشهور است، نام (objectName) این دکمه را به cmdCalculate تغییر می‌دهیم. اکنون ابعاد دیالوگ خود را تا حد معقولی کوچک می‌کنیم. سپس در بالای کلید حساب کردن، یک جعبه متن قرار می‌دهیم (Line Edit) نام آنرا نیز به txtInput تغییر می‌دهیم. دقت کنید که نامی که در اینجا برای ویجت‌ها تنظیم می‌کنیم، همان نام متغیریست که در کد با استفاده از آن به این ویجت دسترسی پیدا می‌کنیم. از این روست که نام‌های معقول و مرتبط برای هر ویجت تنظیم می‌کنیم.

در مرحله بعدی می‌خواهیم برای رخداد کلیک شدن بر روی دکمه، کدی بنویسیم. روی دکمه



راست کلیک می‌کنیم و از منوی ظاهر شده Go to slot را انتخاب می‌کنیم. از لیست ظاهر شده عبارت clicked که مربوط به رخداد کلیک شدن است انتخاب می‌کنیم. Creator در فایل dialog.h ما پروتوتایپ تابع کلیک شدن و در فایل dialog.cpp ما بدنه آنرا ایجاد می‌کند و آنرا باز کرده، روبروی شما برای کد زدن قرار می‌دهد. اگر دقت کنید متوجه می‌شوید که تابع نام خاصی دارد (on_cmdCalculate_clicked) نامهایی به شکل on_objectName_slot که برای توابع استفاده شوند به صورت خودکار توسط پیش‌پردازنده کیوت به شیار مربوطه متصل می‌شوند. قبل از اینکه به کد زدن در داخل این تابع بپردازیم، چند مفهوم را بررسی می‌کنیم.

Signals & Slots

الگو و تکنولوژی سیگنالها و شیارها در کیوت به عنوان یک پیاده‌سازی بسیار موفق Observer Pattern برای اولین بار در محیط‌های برنامه‌سازی حرفه‌ای به کار گرفته شده است. در این تکنولوژی هر شیء در برنامه می‌تواند یک سیگنال تولید کند و هر شیء نیز تعدادی شیار دارد که در صورت رخداد سیگنال خاصی اجرا می‌شود. قابل توجه است که می‌توان یک یا چند سیگنال را به یک یا چند شیار متصل (connect) نمود.

در برنامه‌نویسی رخدادگرا، معمولا سیگنال‌هایی که سیستم به برنامه ارسال می‌کند در قالب یک رخداد ترجمه شده، توسط یک تابع دریافت و انجام می‌شود (مانند کلیک بر روی یک دکمه). مشکلی که در این سیستم وجود دارد آنست که رخدادها از پیش تعریف شده و ثابت هستند و هر رخدادی فقط یک کار انجام می‌دهد. بنابراین اگر شما بخواهید رخدادی تهیه کنید که با کلیک بر روی یک دکمه، برنامه بسته شود، باید در تابعی مربوط به رخداد کلیک، تابعی مربوط به بسته شدن برنامه را فراخوانی کنید.

اما در مدل سیگنال‌ها و شیارها، کافیست سیگنال کلیک شدن بر روی کلید را به شیار بسته شدن پنجره متصل کنید. (از طریق Designer و کلید F4 می‌توانید اینکار را تصویری انجام دهید.) همچنین مدل سیگنال‌ها و شیارها برای برنامه‌سازی ناهمگون (Asynchronous Programming) مانند برنامه‌سازی شبکه بسیار کارآمد هستند.

از آنجایی که در کیوت این مدل توصیه می‌شود و وجود دارد، در مواردی که رخدادها با سیگنال‌ها و شیارها همپوشانی داشته‌اند، مدل رخدادها حذف شده و سیگنال و شیار جایگزین آن شده است. به همین دلیل کلیک بر روی یک دکمه با قالب سیگنال و شیار مدیریت می‌شود، اما مواردی مانند استفاده از کیبورد یا رسم پنجره که رخدادهای پایه‌ای سیستم عامل‌ها هستند هنوز با مدل رخدادی مدیریت می‌شوند. در ادامه این راهنما بیشتر راجع به رخدادها و استفاده از آنها بحث می‌کنیم.

مدیریت و کد شیار

در شیار کلیک شدن بر روی دکمه خود می‌خواهیم کدی بزنیم که محتوی txtInput را دریافت کند، آنرا به عدد تبدیل کند (چون محتوی یک textbox , line edit قطعا یک string است)، سپس رادیکال عدد را محاسبه کند و در یک MessageBox آنرا نمایش دهد.

قبل از آنکه فرآیند نوشتن کد را شروع کنیم مفهومی دیگر را بررسی می‌کنیم.

ویژگیها و صفات (Properties)

در مدل برنامه‌نویسی GUI اشیایی داریم که ظاهر تصویری دارند. یک دیالوگ، پنجره، دکمه، ورودی متن، لیست و هر چیز دیگری که روی یک پنجره قرار می‌گیرد یک شیء است که تفاوت اندکی با اشیاء معمولی دارد. اینگونه اشیاء را Control یا Widget (یا گاها Component, OCX, ActiveX) می‌نامند. هرویجت مانند هرشیء دیگر، دارای دوچیز است: توابع و متغیر. متغیرهای هر شیء داده‌های آنرا نگهداری می‌کنند. به عنوان مثال متغیر m_text در یک دکمه، متن روی دکمه را نگهداری می‌کند. توابع هر شیء معمولا کاری مربوط به آن شیء انجام می‌دهند. به عنوان مثال تابع setText متن دکمه را تغییر می‌دهد و تابع hide دکمه را مخفی می‌کند (غیب می‌شود).

از آنجایی که متغیرهای یک شیء نباید تنها تغییر کنند ـ به عنوان مثال اگر متغیر m_visible مربوط به نمایان بودن یا مخفی بودن یک دکمه را دستی false کنیم، دکمه مخفی نمی‌شود. بلکه باید آنرا false کنیم و یکبار دیگر دکمه را رسم کنیم ـ مفهومی به نام Property وجود دارد. هر متغیری از یک شیء که لازم است پس از تغییر آن، مقداری کد نیز جهت تنظیمات اجرا شود، Property نامیده

می‌شود. برای اینکه برنامه‌نویس نتواند Property ها را دستی تغییر دهد و کنترل برنامه را بهم بریزد، Property ها مستقیما قابل تغییر نیستند. برای تغییر هر متغیری که Property است، باید از تابع استفاده کرد. بنابراین شما اصلا متغیر m_text مربوط به txtInput را نمی‌بینید، بلکه با تابع setText مقدار آنرا تغییر می‌دهید و اگر می‌خواستید مقدار آنرا بدانید یا جایی استفاده کنید، بازهم از تابع text استفاده می‌کنید و هیچگاه مستقیما به m_text دسترسی ندارد. ساده‌ترین کدی که این دو تابع (که با نام‌های Property Setter و Property Getter شناخته می‌شوند) می‌توانند داشته باشند به صورت زیر است:

```
QString text()
{
	return m_text;
}
void setText(Qstring newText)
{
	m_text=newText;
}
```

شاید به نظر برسد وجود این دو تابع معقول نیست و کارکردن مستقیم با متغیر ساده‌تر است، اما معمولا توابع get و set دارای کدهای دیگری جهت کنترل برنامه هستند (مثلا hide نه تنها m_visible را false می‌کند، بلکه دکمه را یکبار دیگر نیز رسم می‌کند)

بنابراین در یک شیء گرافیکی و سطح بالا معمولا هیچ متغیری نداریم که توسط برنامه‌نویس قابل دسترسی باشد، و تقریبا همه کارها توسط این دو نوع تابع انجام می‌گیرند.

ادامه پروژه

اکنون کد تابع را قدم به قدم می‌نویسیم و بررسی می‌کنیم:

```
QString strNum = ui->txtInput->text();
```

از آنجایی که txtInput یک عضو دیالوگ تصویری ما، یعنی متغیر ui بود، لازم است تا برای دسترسی به آن از ui گذر کنیم. از آنجایی که تابع text مربوط به textInput ما یک تابع عضو آن است، از داخل آن دسترسی می‌گیریم. از آنجایی که نوع خروجی این تابع یک رشته است، یک رشته کیوت تعریف می‌کنیم و نوع خروجی را درون آن قرار می‌دهیم.

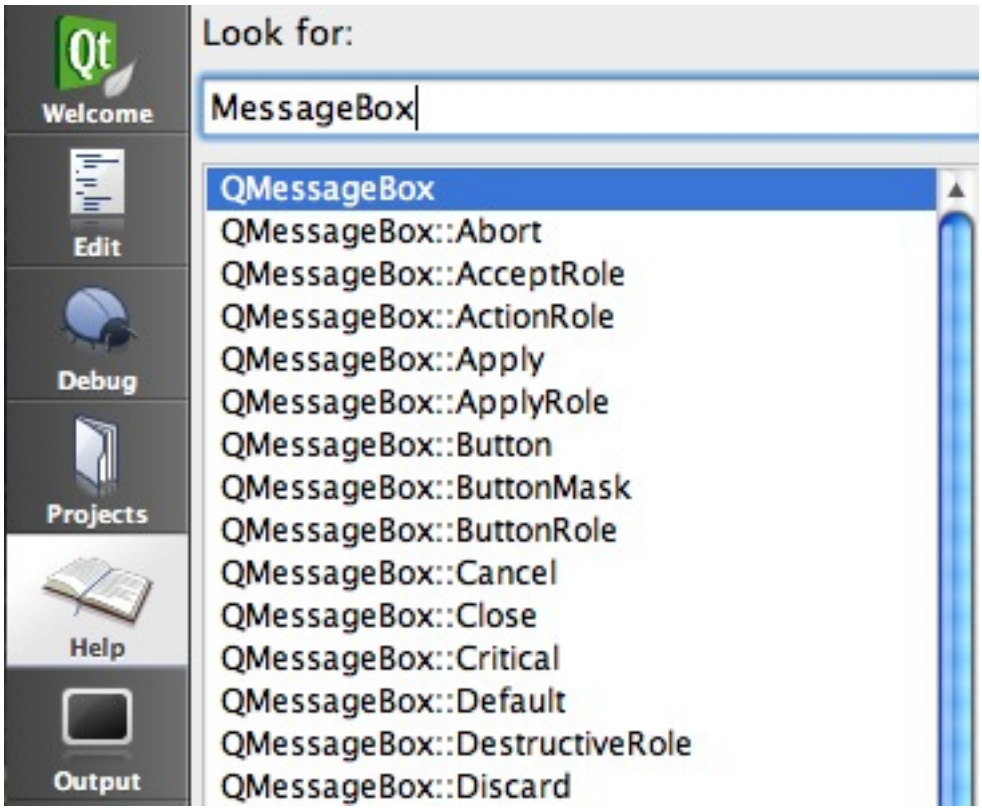
```
int intNum=strNum.toInt();
```

از آنجایی که strNum یک عدد در قالب رشته بود، یعنی مثلا "۱۲۳" بود نه ۱۲۳، باید آنرا به عدد تبدیل کنیم. خوشبختانه چون اینکار فرآیندی متداول است، کد آن داخل تابع toInt عضو strNum وجود دارد. این تابع معادل عددی یک رشته را بر می‌گرداند. ما هم آنرا داخل متغر int خود قرار داده‌ایم.

در ادامه از تابع sqrt داخل کتابخانه math.h استفاده می‌کنیم و این کتابخانه را در بالای فایل ضمیمه می‌کنیم:

```
int intNumSquared=sqrt(intNum);
```

اکنون می‌خواهیم از MessageBox که یک ساختار معمول در برنامه‌های GUI است استفاده کنیم. جعبه‌پیام همان دیالوگ بسیار آشناییست که خطاها و پیام‌های برنامه‌ها را نمایش می‌دهد و معمولا دکمه Ok یا Yes , No و از این قبیل دکمه‌ها با یک پیام چند خطی و شمایلی آشنا و یکسان در همه برنامه‌ها دارد. از آنجایی که نمی‌دانیم چگونه در داخل کیوت از این ساختار استفاده کنیم، به راهنمای کیوت سر می‌زنیم (Help در لیست سمت چپ)



عبارت MessageBox را داخل فیلد جستجو تایپ می‌کنیم. QMessageBox نام شیئیست که کیوت برای کارکردن با MessageBox در اختیار ما قرار می‌دهد. اکنون دو راه داریم، یا کلید Enter را می‌زنیم و توضیحات مربوط به QMessageBox را مطالعه کرده، یاد می‌گیریم و یا حدس می‌زنیم که QMessageBox::information چیزیست که ما به دنبال آن هستیم! چیزی که راهنما به ما نمایش می‌دهد، پروتوتایپ این تابع است:

```
StandardButton  information ( QWidget * parent, const QString & title, const QString & text, StandardButtons buttons = Ok, StandardButton defaultButton = NoButton )
```

از پروتوتایپ متوجه می‌شویم که تابع سه ورودی اجباری و دو ورودی اختیاری دارد. ورودی اول یک ویجت است که از نام آن می‌فهمیم پدر جعبه پیام، یعنی پنجره ماست. پارامتر دوم عنوان جعبه و سومی متن جعبه است. اکنون از این تابع استفاده می‌کنیم:

QMessageBox::information(this , "This is the title!" , QString("The square root of ١٪ is ٢٪").arg(intNum).arg(intNumSquared));

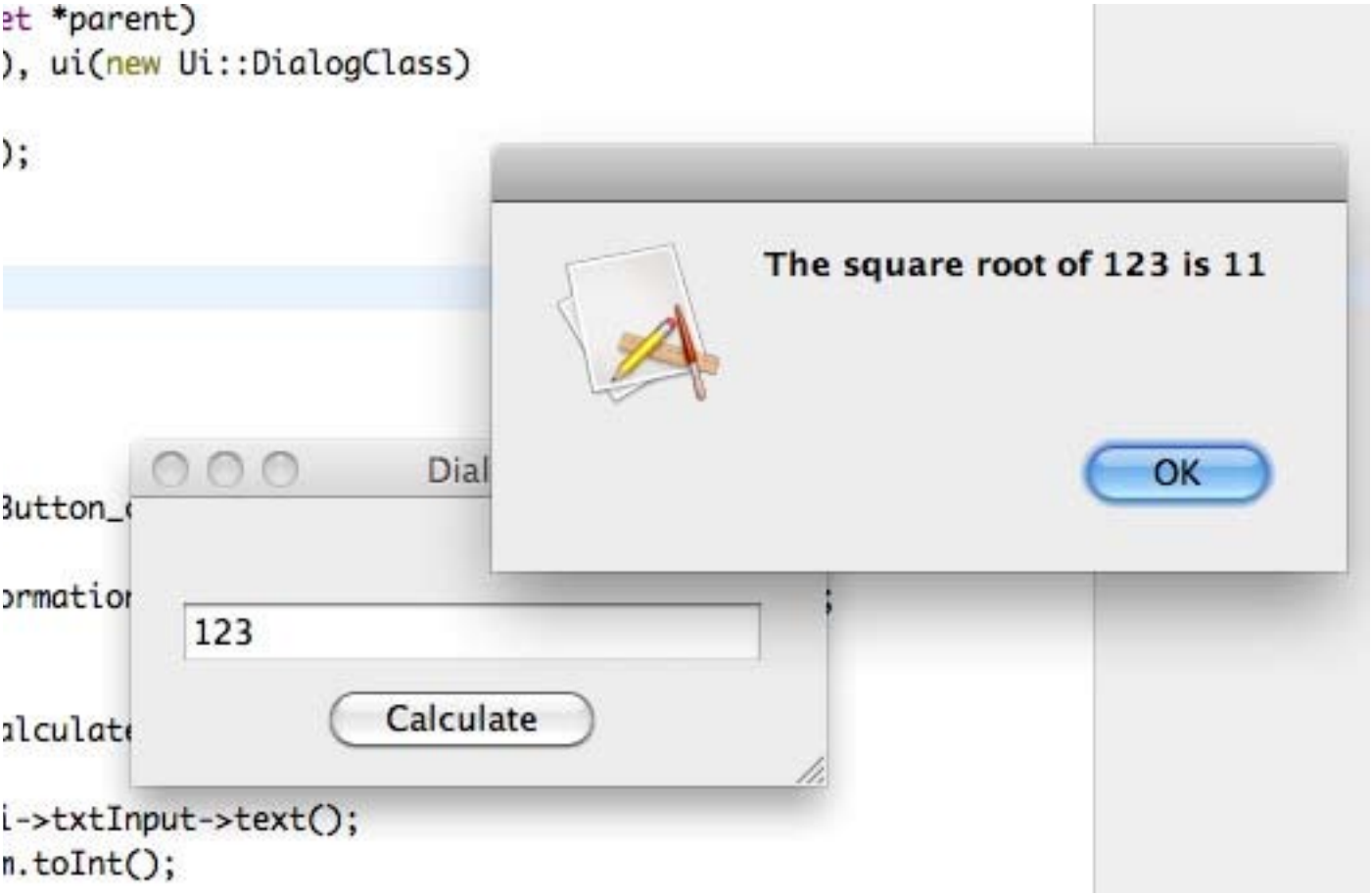
پارامتر اول، یعنی this یک اشاره‌گر به دیالوگیست که داخل آن در حال کد زدن هستیم. این پارامتر مشخص می‌کند که جعبه پیام باید روی دیالوگ ما باز شود و تا وقتی بسته نشده دیالوگ قفل باشد. پارامتر دوم، عنوان پنجره جعبه است، یعنی چیزی که در قسمت آبی بالای پنجره نوشته می‌شود. پارامتر سوم، متنیست که جعبه برای ما نمایش می‌دهد. از اینجایی که این متن باید رشته‌ای از نوع QString باشد ولی مقادیری که ما قصد نمایششان را داریم، int هستند، از امکانات QString بهره می‌جوییم. دقت کنید که در غیر این صورت مجبور بودیم مجددا دو پارامتر خود را به QString تبدیل کنیم و از آنها استفاده کنیم. هر تعداد پارامتری که در متن می‌خواهیم، مانند printf، با درصد و یک شماره (شماره پارامتر) مشخص می‌کنیم. سپس با تابع arg این آرگومان‌ها را تنظیم می‌کنیم. تابع arg ۲۰ بار overload شده و همه نوع متغیر را دریافت می‌کند.

در انتها دو فایل math.h و QMessageBox را ضمیمه می‌کنیم. کل کد فایل dialog.cpp به صورت زیر در می‌آید:

```
#include "dialog.h"
#include "ui_dialog.h"
#include <math.h>
#include <QMessageBox>
Dialog::Dialog(QWidget *parent)
    : QDialog(parent), ui(new Ui::DialogClass)
{
    ui->setupUi(this);
}

Dialog::~Dialog()
{
    delete ui;
}

void Dialog::on_pushButton_clicked()
{
    QMessageBox::information(this,tr("Title"),tr("Hello there!"));
}

void Dialog::on_cmdCalculate_clicked()
{
```

```
    QString strNum=ui->txtInput->text();
    int intNum=strNum.toInt();
    int intNumSquared=sqrt(intNum);
    QMessageBox::information( this , "This is the title!" , QString("The square root of ١٪ is ٢٪") .arg(intNum).arg(intNumSquared));

}
```

برنامه را ذخیره می‌کنیم و اجرا می‌کنیم.



در ادامه قصد داریم برای آشنا شدن با دیگر ساختارهای برنامه کیوت و رخدادها، برنامه را طوری تغییر دهیم که اگر کاربر کلید Escape را بر روی کیبورد فشرد، به جای خروج، عدد موجود در فیلد متنی دوبرابر شود. مجددا به Help سر زده و عبارت keypress را جستجو می‌کنیم. از لیست عبارت keyPressEvent را انتخاب می‌کنیم. لیستی از این رخداد باز می‌شود که ویجت‌های مختلف را در بر دارد. QDialog که پنجره ماست، مورد نظر است. به ما اطلاع می‌دهد که همان QWidget است، بنابراین توضیحات مربوط به آنرا مطالعه می‌کنیم:

void QWidget::keyPressEvent (QKeyEvent * event) [virtual protected]

توضیحات را مطالعه می‌کنیم و متوجه می‌شویم اطلاعات کلید فشرده شده درون event قرار می‌گیرد

که از نوع QKeyEvent است. برای آشنایی بیشتر با این نوع روی آن کلیک می‌کنیم. پس از خواندن توضیحات متوجه می‌شویم که تابع key نوع کلید فشرده شده را برمی‌گرداند و انواع مختلف کلید در Qt::Key قرار دارد. بنابراین کد را به صورت زیر در dialog.cpp می‌نویسیم:

```
void Dialog::keyPressEvent(QKeyEvent *e)
{
   if (e->key()==Qt::Key_Escape)
   {
      int intNum=ui->txtInput->text().toInt();
      ui->txtInput->setText(QString("۱٪").arg(intNum*۲));
   }
}
```

همچنین علاوه بر ضمیمه کردن فایل QKeyEvent، خط زیر را نیز در dialog.h می‌افزاییم:

```
void keyPressEvent(QKeyEvent *);
```

از آنجایی که نکته مبهمی در کد وجود ندارد از توضیح آن اجتناب می‌کنیم. اکنون برنامه را اجرا می‌کنیم و می‌بینیم که صحیح کار می‌کند.

نکات انتهایی

کیوت یک محیط بسیار مجهز و حرفه‌ایست که برای استفاده از هریک از امکانات منحصر بفرد آن دانش خاصی لازم است. خوشبختانه کیوت مجموعه کاملی از راهنما و آموزش را در Help خود جمع آوری کرده است. چندین کتاب هم در رابطه با کیوت وجود دارد که هرکدام جنبه‌های خاصی از این چهارچوب را بررسی می‌کنند. در حالت کلی، بهترین راه برای یادگیری محیط‌های مثل کیوت، کار تیمی، سعی و خطا، مطالعه و تلاش است.