This is an explanation of a Qt/C++ program written with Qt Creator IDE to draw a line or rectangle dynamically on a QWidget.

I start from the basics as this is my second write up on Qt/C++. Here first a **linewidget class** is created by subclassing from **QWidget**. Also concurrently Qt Creator generates a file named **linewidget.ui** to assist in developing GUI visually and easily. But when we add or change this file there must be a way to reflect that in our **linewidget.cpp** file.

It is done in this way: **linewidget.ui** is infact an **xml** file. QtCreator generates a class file out of this xml file named **ui_linewidget.h**. Let's have a look into this file below, it is generally found in our project build directory. At the end of this file we see that our **linewidget** class is actually inheriting from this autogenerated class **ui_linewidget.h** and it is done in a namespace name **Ui** for convenience:

```
/********************************************************************
***********
** Form generated from reading UI file 'linewidget.ui'
**
** Created by: Qt User Interface Compiler version 5.4.1
**
** WARNING! All changes made in this file will be lost when
recompiling UI file!
********************************************************************
**********/

#ifndef UI_LINEWIDGET_H
#define UI_LINEWIDGET_H

#include <QtCore/QVariant>
#include <QtWidgets/QAction>
#include <QtWidgets/QApplication>
#include <QtWidgets/QButtonGroup>
#include <QtWidgets/QHeaderView>
#include <QtWidgets/QPushButton>
#include <QtWidgets/QVBoxLayout>
#include <QtWidgets/QWidget>

QT_BEGIN_NAMESPACE
```

```
class Ui_lineWidget
{
public:
    QWidget *widget;
    QVBoxLayout *verticalLayout;
    QPushButton *btnLine;
    QPushButton *btnRect;

    void setupUi(QWidget *lineWidget)
    {
        if (lineWidget->objectName().isEmpty())
            lineWidget->setObjectName(QStringLiteral("lineWidget"));
        lineWidget->resize(517, 336);
        widget = new QWidget(lineWidget);
        widget->setObjectName(QStringLiteral("widget"));
        widget->setGeometry(QRect(410, 10, 77, 54));
        verticalLayout = new QVBoxLayout(widget);
        verticalLayout->setSpacing(6);
        verticalLayout->setContentsMargins(11, 11, 11, 11);
verticalLayout->setObjectName(QStringLiteral("verticalLayout"));
        verticalLayout->setContentsMargins(0, 0, 0, 0);
        btnLine = new QPushButton(widget);
        btnLine->setObjectName(QStringLiteral("btnLine"));

        verticalLayout->addWidget(btnLine);

        btnRect = new QPushButton(widget);
        btnRect->setObjectName(QStringLiteral("btnRect"));

        verticalLayout->addWidget(btnRect);


        retranslateUi(lineWidget);

        QMetaObject::connectSlotsByName(lineWidget);
    } // setupUi

    void retranslateUi(QWidget *lineWidget)
```

```
    {
lineWidget->setWindowTitle(QApplication::translate("lineWidget",
"lineWidget", 0));
        btnLine->setText(QApplication::translate("lineWidget", "Line",
0));
        btnRect->setText(QApplication::translate("lineWidget",
"Rectangle", 0));
    } // retranslateUi

};

namespace Ui {
    class lineWidget: public Ui_lineWidget {};
} // namespace Ui

QT_END_NAMESPACE

#endif // UI_LINEWIDGET_H
```

Now let's have a look at our **linewidget.h** file, we shall see it declares here the namespace and a **forward declaration** of the class **linewidget**. Thus our this class inherits all the things from the **ui_linewidget** class plus here again we inherit from the **QWidget** to get all the properties of it as well.

Here **#pragma** once is a Qt macro that avoids multiple inclusion of this header file. And **Q_OBJECT** is another macro which tells the Qt meta object compiler (moc) that this class inherits from QObject as well. Hence our linewidget class is inheriting like this -> **linewidget:QWidget:QObject**. Also this macro gives the background supports for the Qt **slot and signal** mechanism which is not part of the original C++. Other than that some specific comments are added in this code segment:

```
#pragma once

#include <QPainter>
#include <QWidget>

namespace Ui {
    class lineWidget;
}
```

```
class lineWidget : public QWidget
{
    Q_OBJECT

public:
    //the constructor is marked explicit so that we don't get any
    // implicit conversion by passing the wrong parameter
    explicit lineWidget(QWidget *parent = 0); //also it has a default
null pointer value, so a linewidget can be created with null parameter
(i.e when we don't specify the constructor parenthesis at all!)
    bool mousePressed;
    bool drawStarted;
    int selectedTool;
    //destructor is needed when we construct a object on the heap
instead of stack
    // for efficient memory management
    ~lineWidget();

protected:
    void mousePressEvent(QMouseEvent *event);
    void mouseMoveEvent(QMouseEvent *event);
    void mouseReleaseEvent(QMouseEvent *event);
    void paintEvent(QPaintEvent *event);

private slots:
    void on_btnLine_clicked();
    void on_btnRect_clicked();

private:
    //we declare a linewidget pointer object using namespace
identifier;
    Ui::lineWidget *ui;
    QPainter painter;
    QPixmap mPix;
    QLine mLine;
    QRect mRect;

};
```

Here is the **linewidget.cpp** file, which in the constructor takes a pointer to parent as argument. Also in the header file we said that it could be 0 as well. Let's see how we are actually creating a object of **linewidget** type in **main.cpp** file, we are just creating a object without any parameter i.e we are calling the default constructor i.e it has no parent i.e it is the last element in the hierarchy chain. Also we see that nowhere in the linewidget file we declared **show()** method but we can use it as it is inherited from **QWidget** class:

```
    lineWidget w;
    w.show()
```

```
#include "linewidget.h"
#include "ui_linewidget.h"

#include <QMouseEvent>
#include <QPainter>

lineWidget::lineWidget(QWidget *parent) :
    QWidget(parent),
    ui(new Ui::lineWidget)
{
    ui->setupUi(this);
    mPix = QPixmap(400,400);
    mPix.fill(Qt::white);

    //set everything to false as nothing has started yet
    mousePressed = false;
    drawStarted = false;

    //default is line
    selectedTool = 2;
}


void lineWidget::mousePressEvent(QMouseEvent* event){
    //Mouse is pressed for the first time
    mousePressed = true;
```

```
    //set the initial line points, both are same
    if(selectedTool == 1){
        mRect.setTopLeft(event->pos());
        mRect.setBottomRight(event->pos());
    }
    else if (selectedTool == 2){
        mLine.setP1(event->pos());
        mLine.setP2(event->pos());
    }
}

void lineWidget::mouseMoveEvent(QMouseEvent* event){

    //As mouse is moving set the second point again and again
    // and update continuously
    if(event->type() == QEvent::MouseMove){
        if(selectedTool == 1){
            mRect.setBottomRight(event->pos());
        }
        else if (selectedTool == 2){
            mLine.setP2(event->pos());
        }
    }

    //it calls the paintEven() function continuously
    update();
}

void lineWidget::mouseReleaseEvent(QMouseEvent *event){

    //When mouse is released update for the one last time
    mousePressed = false;
    update();
}

void lineWidget::paintEvent(QPaintEvent *event){

    painter.begin(this);
```

```
    //When the mouse is pressed
    if(mousePressed){
        // we are taking QPixmap reference again and again
        //on mouse move and drawing a line again and again
        //hence the painter view has a feeling of dynamic drawing
        painter.drawPixmap(0,0,mPix);
        if(selectedTool == 1)
            painter.drawRect(mRect);
        else if(selectedTool == 2)
            painter.drawLine(mLine);

        drawStarted = true;
    }
    else if (drawStarted){
        // It created a QPainter object by taking  a reference
        // to the QPixmap object created earlier, then draws a line
        // using that object, then sets the earlier painter object
        // with the newly modified QPixmap object
        QPainter tempPainter(&mPix);
        if(selectedTool == 1)
            tempPainter.drawRect(mRect);
        else if(selectedTool == 2)
            tempPainter.drawLine(mLine);

        painter.drawPixmap(0,0,mPix);
    }

    painter.end();
}

lineWidget::~lineWidget()
{
    delete ui;
}
//only two button is on the ui btnLine and btnRect
void lineWidget::on_btnLine_clicked()
{
    selectedTool = 2;
```

```
}

void lineWidget::on_btnRect_clicked()
{
    selectedTool = 1;
}
```

We use ” ” to identify local header file and < > to indicate library header file located in specific location for that specific platform.

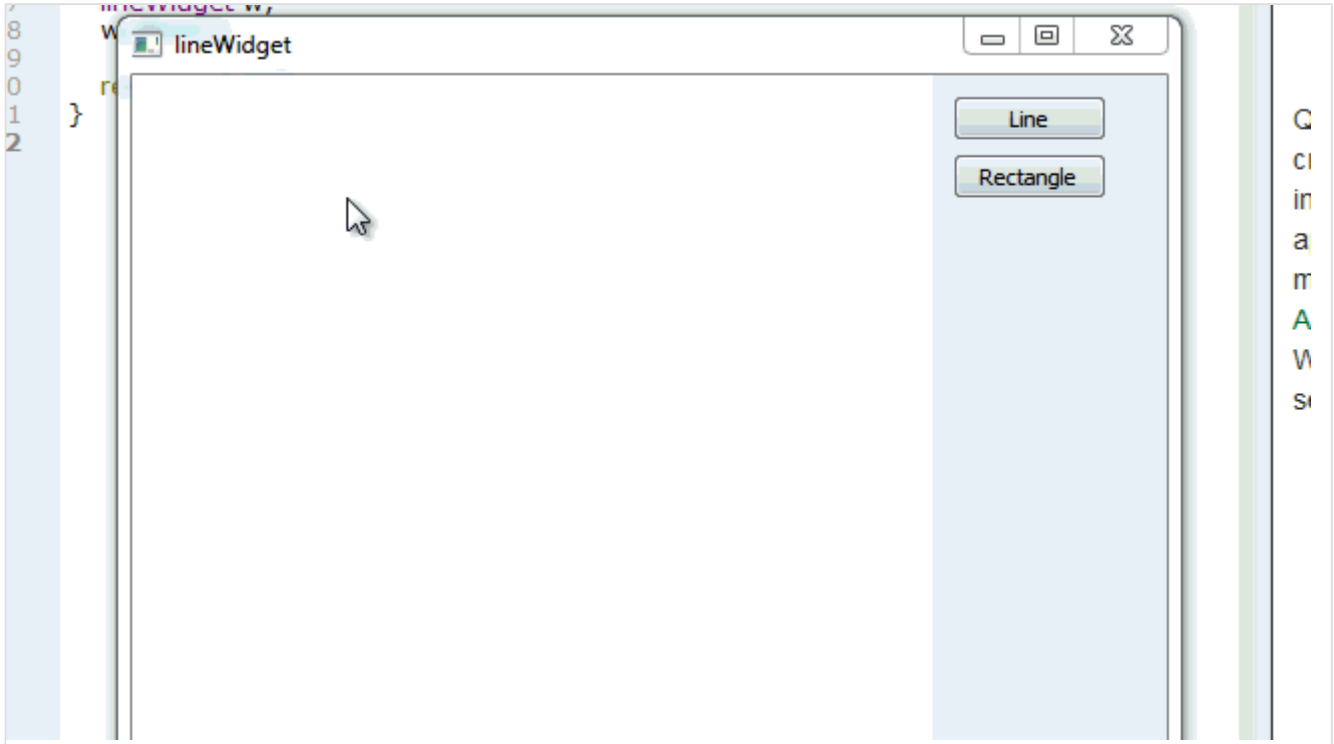
Now, how the line or rectangle is drawn on the QtWidget is mostly commented in the source code. Basic principle is when the mouse is clicked two points are recorded. On mouse move events we take the second point continuously and update on a **QPixmap** which has been used to construct a **QPainter**. Later when the mouse button is released we draw the shape on this **mPix** reference. Which is finally set on the earlier **painter** object. Thus newly created items stays. This is the main.cpp file just in case:

```
#include "linewidget.h"
#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    lineWidget w;
    w.show();

    return a.exec();
}
```

Here is a demo of the drawing:



line and rectangle drawing