GUI Applications with C++
Build GUI apps for linux with QT

Nathan Warner

Northern Illinois
University

Computer Science
Northern Illinois University
November 23, 2023
United States

Contents

Preface

Creating a simple GUI application with Qt in C++ involves several steps. First, ensure you have the Qt framework installed on your Linux system. You can download QT on arch linux using pacman.

```
pacman -Sy qt5
```

Source File (Simple GUI)

Here we make a simple GUI to show how the projects are built

```
#include <QApplication>
#include <QProcess>
#include <QWidget>
#include <QPushButton>
#include <iostream>

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);

    QWidget window;

    QPushButton *button = new QPushButton("Click me", &window);
    button->setGeometry(50, 50, 80, 30);

    window.resize(250, 150);
    window.setWindowTitle("Button Example");
    window.show();

    return app.exec();
}
```

.pro file (Making project)

To build our projects, we need to assemble a .pro file. A sample .pro file for the file above would look something like

```
TEMPLATE = app
TARGET = your_app_name
CONFIG += console c++11
QT += widgets

# Input
SOURCES += main.cpp

# If you have additional source files, list them here
# SOURCES += source1.cpp source2.cpp

# If you have header files, list them here
# HEADERS += header1.h header2.h

# If you have UI files created using Qt Designer, list them here
# FORMS += mainwindow.ui
```

- **TEMPLATE=app** tells Qt's build system, qmake, that your project is an application. This means qmake will generate a Makefile to build an executable.
- **TARGET=you_app_name** specifies the name for the executable
- **CONFIG += console c++11** is used to define various configuration options for the build process
  - **console:** This option is used to specify that the application is a console application. This is particularly relevant on Windows, where it determines whether a console window is opened alongside your application. For GUI applications, you typically wouldn't include console, as GUI applications on Windows usually don't need a console window. On Linux and macOS, the distinction is less significant, as terminal-based and graphical applications are not as strictly separated as on Windows.
  - **c++11** tells the compiler to use the C++11 standard
- **QT += widgets** is used to add the widgets module form the list of modules to be included in the application
- **SOURCES += main.cpp** is used to add source files in the build process

Building the application

Once we have made the .pro file, we can begin the build process

```
qmake filename.pro # First
make # Then
```

Other options for pro file variables

- **TEMPLATE:**
  - **app:** This is used for building an application. It will create an executable. When you're developing a typical GUI or console application, you use TEMPLATE = app.
  - **lib:** This is used for building a library. If you're developing a library (either static or dynamic), you would use TEMPLATE = lib. When you choose this, qmake will generate a Makefile suitable for building a library instead of an executable.
  - **subdirs:** This is used for a project that contains multiple subprojects. The subdirs template is useful when your project is large and split across multiple directories, each of which is a different project (either an application or a library). With TEMPLATE = subdirs, qmake will manage these subprojects according to the instructions you provide in the .pro file.
  - **aux:** This template is used for auxiliary files that are not compiled but are included in the project. It's less commonly used compared to the other templates.
  - **vcapp and vclib:** These are for Visual Studio integration on Windows.
- **CONFIG:**
  - **debug:** Builds the application with debugging symbols. Useful for debugging the application.
  - **release:** Builds the application in release mode, which typically includes optimizations and lacks debugging symbols.
  - **qml_debug:** Enables debugging of QML code. Useful if you are using QML for your application's UI.
  - **qt:** Ensures that Qt-specific build steps are executed, like running the Meta-Object Compiler (MOC) on classes that use Qt's signal and slot mechanism.
  - **c++11, c++14, c++17, c++20:** Specifies the C++ standard to be used. You should match this with the version of C++ your code is written in.
  - **warn_on:** Enables all compiler warnings. This is useful for ensuring that potential code issues are highlighted during the build process.
  - **static:** When building a library, this option specifies that the library should be static rather than dynamic.
  - **shared:** Opposite of static, this is used for building shared libraries.
  - **testcase:** Used when building a project as a test case using Qt Test.
  - **lex yacc:** Includes support for Lex and Yacc if you are using these tools in your project.
  - **thread:** Enables multi-threading support in your application.
  - **exceptions:** Enables support for C++ exceptions.
  - **no_keywords:** Disables the use of Qt-specific keywords like slots, signals, and emit to avoid conflicts with third-party libraries.
  - **opengl:** Enables the use of OpenGL in the application.
  - **widgets:** Includes the QtWidgets module, necessary for applications using Qt Widgets.
  - **network:** Includes the QtNetwork module for network programming.
  - **sql:** Includes the Qt SQL module for database operations.

- **xml:** Includes the Qt XML module for XML processing.
- **link_pkgconfig:** Allows the use of pkg-config to find libraries.
- **precompile_header:** Enables the use of precompiled headers to speed up compilation.

- **QT: (List of modules we can add or remove)**
  - **core:** This is included by default and provides core non-GUI functionality. It's essential for any Qt application.
  - **gui:** Also included by default in most cases, this module is necessary for any application that uses Qt's graphical user interface elements.
  - **widgets:** For applications that use QWidget-based user interfaces. This is a key module for traditional desktop GUI applications.
  - **network:** Adds network communication capabilities, like handling TCP/IP connections, for your application.
  - **sql:** If your application needs to interact with SQL databases, this module provides the necessary functionality.
  - **qml:** Necessary for applications that use the QML language for designing user interfaces, especially in combination with the Qt Quick module.
  - **quick:** Used in conjunction with QML to create fluid, dynamic user interfaces. It's a part of the Qt Quick framework.
  - **multimedia:** For applications that need to handle audio, video, radio, and camera functionality.
  - **bluetooth:** Provides classes for writing Bluetooth applications.
  - **concurrent:** Enables easier use of multi-threading in applications.
  - **printsupport:** For applications that require printing capabilities.
  - **webkit or webengine:** For applications that need to embed web content using WebKit or Qt WebEngine.
  - **xml:** Adds support for reading and writing XML data.
  - **opengl:** For applications that use OpenGL for rendering graphics.
  - **testlib:** For writing unit tests.
  - **positioning:** For applications that require location and positioning functionality.
  - **sensors:** Access to various hardware sensors like accelerometers, gyroscopes, etc.
  - **serialport:** For applications that communicate with devices over serial ports.
  - **svg:** Support for Scalable Vector Graphics (SVG) files.
  - **dbus:** For applications that communicate with other applications using the D-Bus protocol (mostly relevant on Linux).

Includes

The main include we need to build our GUI apps is <QApplication>, this class manages the GUI application's control flow and main settings

The <QWidget> class is the base class of all user interface objects.

The <QProcess> allows us to launch external processes

Example Application

Here is a sample application which creates a clickable button that takes us to a certain directory in a new terminal window

```cpp
#include <QApplication>
#include <QWidget>
#include <QPushButton>
#include <QProcess>
#include <iostream>

class MyWindow : public QWidget {
    Q_OBJECT

public:
    MyWindow(QWidget *parent = nullptr) : QWidget(parent) {
        QPushButton *button = new QPushButton("Open Terminal", this);
        button->setGeometry(50, 50, 120, 30);
        connect(button, &QPushButton::clicked, this, &MyWindow::onButtonClicked);
    }

public slots:
    void onButtonClicked() {
        QProcess::startDetached("kitty --working-directory=/home/datura/tmp/cpp");
    }
};

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);

    MyWindow window;
    window.resize(250, 150);
    window.setWindowTitle("Button Example");
    window.show();

    return app.exec();
}

#include "main.moc"
```

Q_OBJECT

The Q_OBJECT macro is essential in Qt applications for any class that defines signals or slots, and it enables several key features provided by Qt's meta-object system.

Signals and Slots

Signals and slots are fundamental aspects of Qt and form the basis of its event communication system. They are used for communication between objects and are an integral part of Qt's programming model, especially in GUI applications. Here's a breakdown of what they are and how they work:

9.1 Signals

Definition: A signal is a message sent by an object to indicate that some event has occurred or some state has changed. In Qt, signals are declared in a class, but they are not implemented in the class. They are just emitted (or "fired") when the event they represent occurs.

Usage: Signals are used to broadcast information. Any number of slots can listen and react to a particular signal.

Syntax: In Qt, signals are declared with the signals keyword.

```
signals:
    void mySignal();
```

9.2 Slots

Definition: A slot is a function that is used to receive and respond to a signal. Slots can be normal member functions of a class.

Usage: Slots are used to perform actions in response to the occurrence of the event signified by a signal. A slot does not know if it has received a signal from a signal or directly as a regular function call.

Syntax: Slots are declared with the slots keyword or can be just regular member functions. They can also be private or protected.

```
public slots:
    void mySlot();
```

9.3 Connecting Signals to Slots

Mechanism: In Qt, objects communicate with each other via signals and slots. When a signal is emitted, all connected slots are called.

Dynamic Connection: The connection between signals and slots can be made at runtime using the QObject::connect() function.

```
connect(sender, SIGNAL(mySignal()), receiver, SLOT(mySlot()));
```

The MyWindow Constructor and Widget Hierarchy

In Qt, the concept of parent-child relationships is pivotal for managing widgets (the graphical user interface elements). When you create a custom widget like MyWindow, which inherits from QWidget, you often include a constructor that allows for specifying a parent widget. This relationship is crucial for several aspects of a widget's behavior and lifecycle.

```
MyWindow(QWidget *parent = nullptr) : QWidget(parent) {
    // Constructor body
}
```

- **Optional Parent Argument:** The constructor of MyWindow takes an optional parameter parent, which is a pointer to a QWidget. The default value of this parameter is nullptr.
- **Delegating to QWidget Constructor:** Inside the constructor, we use an initialization list to call the base class QWidget's constructor with the parent argument. This is a common C++ technique for initializing base class members.
- **Behavior Based on parent Argument:**
  - **Top-Level Window:** If nullptr is passed to MyWindow (or no argument is provided), it implies that MyWindow does not have a parent widget. In this case, MyWindow acts as a top-level window. This means it can be a standalone window with its own window decorations (like a title bar, minimize/maximize buttons, etc.).
  - **Child Widget:** If a valid QWidget pointer is passed as the parent, MyWindow becomes a child widget of the specified parent widget. As a child widget, it will be contained within the parent widget's window and subject to its geometry and visibility. Additionally, the parent widget will manage the lifetime of MyWindow, automatically deleting it when the parent widget is destroyed.

This constructor design allows MyWindow to be versatile: it can either be an independent window or part of a larger interface, depending on how it's instantiated. This flexibility is a key feature of Qt's approach to building user interfaces, where the composition of widgets can be dynamically arranged.

Main Semantics

In a Qt application, the main function serves as the entry point and is responsible for setting up and running the application's main event loop

11.1 QApplication Initialization:

```
QApplication app(argc, argv);
```

- QApplication is a class that manages application-wide resources and is necessary for any Qt GUI application.
- It needs to be instantiated at the beginning of main.
- argc and argv are passed to QApplication to handle any command-line arguments that are relevant to Qt applications (like GUI style, plugin paths, etc.).

11.2 Creating a Window

If no custom window class is made, we can create a window with

```
Qwidget window;
window.resize(250, 150);
window.setWindowTitle("Title Text");
Window.show();
```

11.3 Starting the Event Loop:

```
return app.exec();
```

- app.exec() starts the application's event loop. This is a crucial call; it enters the main loop where events (like mouse clicks, keypresses, or custom events) are received and dispatched to the appropriate widgets.
- The event loop continues to run until exit() is called, usually as a response to events like closing the main window.

main.moc

- **What is main.moc?:** The .moc files are generated by the Meta-Object Compiler (MOC) in Qt. MOC is a tool that processes Qt's specific extensions, like the Q_OBJECT macro, signals, and slots. It generates standard C++ code from these extensions, enabling features like signal-slot connections and introspection.
- **Role of main.moc:** Typically, for classes that include the Q_OBJECT macro or define signals and slots, a corresponding .moc file is generated. However, it's unusual to have a main.moc. In standard Qt applications, the main function usually doesn't define a new class and doesn't include Q_OBJECT. If your main.cpp includes definitions of such Qt classes, then main.moc might be generated and should be included at the end of the main.cpp file. This is not common practice and usually indicates that the application structure could be improved by moving Qt class definitions out of main.cpp.

Defining our own window object

```cpp
#include <QApplication>
#include <QWidget>
#include <iostream>
#include <iomanip>

class MainWindow : public QWidget {
private:
    Q_OBJECT;

public:
    MainWindow(QWidget* parent=nullptr) : QWidget(parent) {

    }

};

int main(int argc, char* argv[]) {
    QApplication app(argc, argv);

    MainWindow window;
    window.resize(1080,700);
    window.setWindowTitle("Title Name");
    window.show();

    return app.exec();
}
#include "main.moc"
```

Creating a button

Now that we have most of the jargon out of the way, lets create some stuff.

14.1 Include Necessary Headers

```
#include <QPushButton>
```

14.2 Creation

In our custom window class,

```
MainWindow(QWidget *parent = nullptr) : QWidget(parent) {
    QPushButton *button = new QPushButton("Do Something", this);
    button->setGeometry(50, 50, 120, 30);
}
```

- **QPushButton *button = new QPushButton(”Open Terminal”, this):** creates a new instance of QPushButton. The text ”Open Terminal” is set as the button’s label. The this pointer is passed as the parent of the button, which means the button is a child widget of MyWindow. As a child widget, it will be displayed within MyWindow and will be deleted when MyWindow is deleted (automatic memory management by Qt).
- **button->setGeometry(50, 50, 120, 30):** sets the position and size of the button within its parent widget (MyWindow). The button is placed at coordinates (50, 50) with a width of 120 pixels and a height of 30 pixels.

14.3 Making the button do something

First, lets create a member function for our window class that provides functionality for our button

```
#include <QProcess>
```

```
public slots:
    void onButtonClick() {
        QProcess::startDetached(
    ↪ "kitty --working-directory=$HOME/tmp/cpp");

    }
```

Then in the constructor we can add the connection

```
connect(object1, signal1, object2, slot2);
connect(button, &QPushButton::clicked, this,
    ↪  &MainWindow::onButtonClicked);
```

- connect(button, &QPushButton::clicked, this, &MainWindow::onButtonClicked); establishes a connection between the clicked signal of the button and the onButtonClicked slot method of MainWindow. This is using the signal-slot mechanism in Qt.
- When the button is clicked, the clicked signal is emitted. Because of the connection established by connect, this will trigger the onButtonClicked method in the MainWindow class.

Args

- **First Argument - button:**
  - This is the source object that emits the signal. In your case, it's the pointer to the QPushButton instance you've created. The signal will be emitted from this button.
- **Second Argument - &QPushButton::clicked:**
  - This specifies the signal you want to connect from the source object. The &QPushButton::clicked is a pointer to the clicked signal of the QPushButton class. This signal is emitted by Qt when the button is clicked.
- **Third Argument - this:**
  - This is the receiver object, which is the object that owns the slot method you want to call. In this case, this refers to the current instance of the MainWindow class, indicating that the slot method belongs to this instance.
- **Fourth Argument - &MainWindow::onButtonClicked:**
  - This specifies the slot, which is the method that will be called in response to the signal. &MainWindow::onButtonClicked is a pointer to the onButtonClicked method of the Wainwindow class. This method should be defined in MainWindow and will be executed when the button is clicked.

Creating Labels

First, we need to make sure that we have <QLabel> included

```
#include <QLabel>
```

Then, in our windows constructor, we can create the label

```
label = new QLabel("Hello, World!", this);
label->setAlignment(Qt::AlignCenter);
label->setGeometry(0, 75, 250, 50); // Adjust geometry as needed
label->hide();
```

QStrings

QString is a fundamental part of the Qt framework, designed to represent strings of text in a way that is optimized for performance and flexibility, especially in the context of internationalization. It's a powerful alternative to standard C++ string types like std::string and C-style strings (char*).

We must use QStrings instead of std::string in our applications to avoid getting errors.

16.1 Creating a QString

Before we create any Qstrings, we must include the header.

```
#include <QString>
```

Then we can create our QString objects

```
QString a = "My QString";
```

16.2 Converting to QString

```
std::string a = "String"
QString b = QString::fromStdString(a); // QString::toStdString()
    ↪ for the reverse
```

16.3 Converting numeric types to QString

```
QString a = QString::number(20);
```

16.4 QStringList

To use QStringList objcets, we first must include the necessary header

```
#include <QStringList>
```

Then to create a QStringList objcet

```
QStringList mylist;
mylist << "item1" << "item2";

list.append("Item 4");
list.insert(2, "Inserted Item");  // Inserts at the specified
↪  index
```

Other useful functions

- **join:** Concatenates all the strings in the list into a single string with a specified separator.
- **sort:** Sorts the list in ascending order.
- **filter:** Returns a new list containing only the strings that match a given pattern.
- **size/count:** Returns the number of items in the list.
- **isEmpty:** Checks if the list is empty.
- **clear:** Clears all items from the list.

Backgrounds

In Qt, there are various ways you can manipulate the background of a widget. These methods offer a range of visual customizations, from simple color changes to more complex patterns and gradients. Here's how you set the background color for the window

```
window.setStyleSheet("background-color: blue;");
widget->setStyleSheet("background-color: blue;"); // More
↪ generally
```

Here's an overview of the different things you can do with backgrounds in Qt:

- **Set a Solid Background Color:**
  - Use setStyleSheet with the background-color property.
  - Example: widget->setStyleSheet("background-color: red;");
- **Use Gradient Backgrounds:**
  - Linear, radial, and conical gradients can be set using style sheets.
  - Example: widget->setStyleSheet("background: qlineargradient(x1:0, y1:0, x2:1, y2:1, stop:0 white, stop:1 black);");
- **Set a Background Image:**
  - Use setStyleSheet with the background-image property.
  - Example: widget->setStyleSheet("background-image: url(:/images/background.png);");
  - You can control how the image is displayed (e.g., stretched, tiled) using background-repeat and background-position.
- **Use a Pixmap as a Background:**
  - This can be done by overriding the paintEvent method in a custom widget.
  - In the paintEvent, use QPainter to draw a QPixmap as the background.
- **Set a Transparent Background:**
  - Useful for creating overlay widgets or for special effects.
  - Example: widget->setStyleSheet("background: transparent;");
- **Apply Box Shadows:**
  - You can use style sheets to add shadows to the background, creating a layer effect.
  - Example: widget->setStyleSheet("box-shadow: 10px 10px 5px #888888;");
- **Use Patterns and Textures:**
  - Similar to using an image, but typically involves tiling a smaller image to create a repeating pattern.
  - Example: widget->setStyleSheet("background-image: url(:/images/pattern.png); background-repeat: repeat;");
- **Adjusting Opacity:**

  - You can adjust the opacity of the background using style sheets.
  - Example: widget->setStyleSheet("background-color: rgba(255, 0, 0, 128);"); (semi-transparent red)

- **Border Customization:**
  - Although not strictly part of the background, borders often complement background customization.
  - Style the borders using properties like border-style, border-width, and border-color.

- **Dynamic Background Changes:**
  - Change backgrounds in response to events like mouse hover, clicks, or other widget states.
  - Use pseudo-states in style sheets (:hover, :pressed, etc.) to achieve this.

17.1 Gradients

Getting the width and height

The QWidget class provides two methods for obtaining these measures **width()** and **height()**

Creating shapes (QPainter)

Creating shapes in a Qt application typically involves using the QPainter class, which provides a rich set of functions to draw various shapes and figures. Here's a guide on how to create different shapes:

18.1 Headers

```
#include <QPainter>      // Used for drawing graphics in widgets
#include <QPoint>        // Represents x and y coordinates in a
↪ 2D space
#include <QRect>         // Defines a rectangle in the plane
↪ using integer precision
#include <QPolygon>      // Represents a polygon defined by a
↪ vector of points
#include <QBrush>        // Used for filling shapes with solid
↪ colors, patterns, or gradients
#include <QPen>          // Used for drawing lines and outlines
↪ of shapes
#include <QImage>        // Represents an image; used in
↪ conjunction with QPainter
#include <QGradient>     // To create gradient objects
```

18.2 Creating a shape

To create shapes, we typically subclass the QWidget class. However, since this document has examples that already have a QWidget derived class (the window class), we use this class insstead. Then we override the paintEvent

```
class MyWidget : public MainWindow {
protected:
    void paintEvent(QPaintEvent *event) override;
};
void MyWidget::paintEvent(QPaintEvent *event) {
    QPainter painter(this);

    // Draw a rectangle
    painter.drawRect(10, 10, 100, 50);

    // Draw a circle
    painter.drawEllipse(10, 70, 50, 50);

    // Draw a line
    painter.drawLine(10, 130, 110, 130);

    // Draw a polygon (triangle in this case)
    QPolygon polygon;
    polygon << QPoint(130, 140) << QPoint(180, 190) <<
↪ QPoint(80, 190);
    painter.drawPolygon(polygon);
}
```

18.3 QPen

Concept 1: **QPen** is used to define the style of the outline of shapes and lines. It primarily sets the color, width, and style of the lines.

```
QPen mypen(Qt::red);
```

- **Color functions:**
  - **setColor(const QColor &color)**: Set the color of the pen.
  - **color()**: Retrieve the current color of the pen.
- **Width functions:**
  - **setWidth(int width)**: Set the width of the pen in pixels.
  - **width()**: Retrieve the current width of the pen.
- **Style functions:**
  - **setStyle(Qt::PenStyle style)**: Set the style of the pen.
  - **style()**: Retrieve the current style of the pen.
- **Cap Style functions:**
  - **setCapStyle(Qt::PenCapStyle capStyle)**: Define the end style of lines.
- **Join Style functions:**
  - **setJoinStyle(Qt::PenJoinStyle joinStyle)**: Set the style of the joins between line segments.
- **Brush functions:**
  - **setBrush(const QBrush &brush)**: Set a brush for the pen.
  - **brush()**: Retrieve the current brush of the pen.
- **Miter Limit functions:**
  - **setMiterLimit(qreal limit)**: Set the miter limit for miter joins.
  - **miterLimit()**: Retrieve the current miter limit.
- **Cosmetic functions:**
  - **setCosmetic(bool cosmetic)**: Set whether the pen is cosmetic.

18.4 QBrush

Concept 2: **QBrush** is used to define the fill pattern of shapes. It can be a solid color, a gradient, or a pattern.

- **Color functions:**
  - **setColor(const QColor &color)**: Set the color of the brush.
  - **color()**: Retrieve the current color of the brush.
- **Style functions:**
  - **setStyle(Qt::BrushStyle style)**: Set the style of the brush.
  - **style()**: Retrieve the current style of the brush.
- **Texture functions:**
  - **setTexture(const QPixmap &pixmap)**: Set a pixmap as a texture for the brush.
  - **setTextureImage(const QImage &image)**: Set an image as the texture for the brush.
- **Gradient functions:**
  - **setGradient(const QGradient &gradient)**: Set a gradient for the brush.
- **Transformation functions:**
  - **setTransform(const QTransform &transform)**: Apply transformations to the brush's pattern.

18.5 Colors for pen and brush

- **Qt::black**: Represents the color black.
- **Qt::white**: Represents the color white.
- **Qt::red**: Represents the color red.
- **Qt::green**: Represents the color green.
- **Qt::blue**: Represents the color blue.
- **Qt::cyan**: Represents the color cyan (a mix of green and blue).
- **Qt::magenta**: Represents the color magenta (a mix of red and blue).
- **Qt::yellow**: Represents the color yellow.
- **Qt::darkRed**: Represents a dark shade of red.
- **Qt::darkGreen**: Represents a dark shade of green.
- **Qt::darkBlue**: Represents a dark shade of blue.
- **Qt::darkCyan**: Represents a dark shade of cyan.
- **Qt::darkMagenta**: Represents a dark shade of magenta.
- **Qt::darkYellow**: Represents a dark shade of yellow.

- **Qt::gray**: Represents the color gray.
- **Qt::darkGray**: Represents a dark shade of gray.
- **Qt::lightGray**: Represents a light shade of gray.
- **Qt::transparent**: Represents a transparent color.

18.6 QPainter methods

- **setPen(const QPen &pen)**: Sets the pen for the painter to *pen*.
- **setBrush(const QBrush &brush)**: Sets the brush for the painter to *brush*.
- **setFont(const QFont &font)**: Sets the font for the painter to *font*.
- **setRenderHint(QPainter::RenderHint hint, bool on = true)**: Sets render hints for the painter.
- **save()**: Saves the current state of the painter.
- **restore()**: Restores the painter to its previously saved state.
- **begin(QPaintDevice *device)**: Begins painting on the specified device.
- **end()**: Ends the painting operation.
- **drawPoint(const QPoint &point)**: Draws a point at the specified position.
- **drawLine(const QPoint &p1, const QPoint &p2)**: Draws a line between two points.
- **drawRect(const QRect &rect)**: Draws a rectangle with the specified dimensions.
- **drawEllipse(const QRect &rect)**: Draws an ellipse within the specified rectangle.
- **drawPolygon(const QPolygon &polygon, Qt::FillRule fillRule = Qt::OddEvenFill)**: Draws a polygon with the specified points and fill rule.
- **drawText(const QRect &rect, int flags, const QString &text)**: Draws the specified text within the given rectangle.
- **drawImage(const QPoint &point, const QImage &image)**: Draws an image at the specified position.
- **drawPixmap(const QRect &rect, const QPixmap &pixmap)**: Draws a pixmap within the specified rectangle.
- **setTransform(const QTransform &transform, bool combine = false)**: Sets a transformation matrix for the painter.
- **translate(const QPointF &offset)**: Translates the coordinate system by the given offset.
- **rotate(qreal angle)**: Rotates the coordinate system by the specified angle.
- **scale(qreal sx, qreal sy)**: Scales the coordinate system by the given factors.
- **setClipRect(const QRect &rect)**: Sets a clipping rectangle for the painter.
- **setOpacity(qreal opacity)**: Sets the opacity level for the painter.

18.7 Shape Objects and their methods

- **QRect**
  - QRect()
  - QRect(const QPoint &topLeft, const QSize &size)
  - QRect(const QPoint &topLeft, const QPoint &bottomRight)
  - QRect(int left, int top, int width, int height)
- **QRectF**
  - QRectF()
  - QRectF(const QPointF &topLeft, const QSizeF &size)
  - QRectF(const QPointF &topLeft, const QPointF &bottomRight)
  - QRectF(qreal left, qreal top, qreal width, qreal height)
- **QPoint**
  - QPoint()
  - QPoint(int xpos, int ypos)
- **QPointF**
  - QPointF()
  - QPointF(qreal xpos, qreal ypos)
  - QPointF(const QPoint &point)
- **QLine**
  - QLine()
  - QLine(const QPoint &p1, const QPoint &p2)
  - QLine(int x1, int y1, int x2, int y2)
- **QLineF**
  - QLineF()
  - QLineF(const QPointF &p1, const QPointF &p2)
  - QLineF(qreal x1, qreal y1, qreal x2, qreal y2)
  - QLineF(const QLine &line)
- **QPolygon**
  - QPolygon()
  - QPolygon(int size)
  - QPolygon(const QPolygon &polygon)
  - QPolygon(const QVector<QPoint> &points)
  - QPolygon(const QRect &rectangle, bool closed = false)
- **QPolygonF**
  - QPolygonF()
  - QPolygonF(int size)
  - QPolygonF(const QPolygonF &polygon)

  - QPolygonF(const QVector<QPointF> &points)
  - QPolygonF(const QRectF &rectangle)

- **QRegion**
  - QRegion()
  - QRegion(const QRegion &region)
  - QRegion(const QRect &rect, QRegion::RegionType type = QRegion::Rectangle)
  - QRegion(int x, int y, int w, int h, QRegion::RegionType type = QRegion::Rectangle)

- **QSize**
  - QSize()
  - QSize(int width, int height)

- **QSizeF**
  - QSizeF()
  - QSizeF(qreal width, qreal height)
  - QSizeF(const QSize &sz)

- **QVector2D**
  - QVector2D()
  - QVector2D(qreal xpos, qreal ypos)
  - QVector2D(const QPoint &point)
  - QVector2D(const QPointF &point)

- **QVector3D**
  - QVector3D()
  - QVector3D(qreal xpos, qreal ypos, qreal zpos)
  - QVector3D(const QPoint &point)
  - QVector3D(const QPointF &point)

- **QVector4D**
  - QVector4D()
  - QVector4D(qreal xpos, qreal ypos, qreal zpos, qreal wpos)
  - QVector4D(const QVector2D &vector)
  - QVector4D(const QVector3D &vector)
  - QVector4D(const QVector3D &vector, qreal wpos)

18.8 Drawing the objects

- **drawLine(const QPoint &p1, const QPoint &p2)**: Draws a line between the points *p1* and *p2*.
- **drawRect(const QRect &rect)**: Draws the outline of a rectangle specified by *rect*.
- **fillRect(const QRect &rect, const QBrush &brush)**: Fills the rectangle *rect* with the brush *brush*.
- **drawEllipse(const QRect &rect)**: Draws an ellipse inscribed in the rectangle *rect*.
- **drawPolygon(const QPolygon &polygon, Qt::FillRule fillRule = Qt::OddEvenFill)**: Draws a polygon defined by *polygon* with the specified fill rule.
- **drawArc(const QRect &rect, int startAngle, int spanAngle)**: Draws an arc defined by the rectangle *rect*, starting at *startAngle* and spanning *spanAngle*.
- **drawPie(const QRect &rect, int startAngle, int spanAngle)**: Draws a pie section defined by the rectangle *rect*, starting at *startAngle* and spanning *spanAngle*.
- **drawChord(const QRect &rect, int startAngle, int spanAngle)**: Draws a chord (a segment of an ellipse) defined by the rectangle *rect*, starting at *startAngle* and spanning *spanAngle*.
- **drawText(const QRect &rect, int flags, const QString &text)**: Draws the text *text* within the rectangle *rect*, using the alignment flags *flags*.
- **drawImage(const QRect &target, const QImage &image, const QRect &source)**: Draws the part of the image *image* specified by *source* into the rectangle *target*.

Note:

Only use one painter object per paintevent

QColor (defining colors)

Concept 3: In Qt, you can use HTML-style color codes with QColor and then set that QColor to a QBrush. HTML-style color codes are typically hex values prefixed with a hash (#). Here's how you can modify your code to use an HTML color for your QBrush:

```
#include <QColor>

QColor mycolor("#808080");
QBrush newbrush(mycolor);
```

19.1 Gradients

19.1.1 Header

```
#include <QGradient>
```

19.1.2 Types of Gradients

- **QLinearGradient(x1,y1,x2,y2)**
  - (x1, y1) and (x2, y2) are the starting and ending points of the gradient line.
- **QRadialGradient(cx, cy, radius, fx, fy)**
  - (cx, cy) is the center of the circle.
  - radius is the radius of the circle.
  - (fx, fy) is the focal point of the gradient; if not set, it defaults to the center.
- **QConicalGradient(cx, cy, startAngl)**
  - (cx, cy) is the center point of the gradient.
  - startAngle is the angle in degrees at which the gradient starts.

19.1.3 setColorAt

To set the colors of the Gradient objects, we use the **setColorAt()** function. This function has the following signature

- **setColorAt(qreal position, const QColor& color)**
  - **position:** A qreal value (a floating-point number) that represents the position along the gradient's axis. For linear and radial gradients, this value is typically between 0.0 and 1.0, where 0.0 represents the start of the gradient and 1.0 represents the end. In a conical gradient, it represents an angle in degrees.
  - **color:** A QColor object representing the color to be used at the specified position.

Stylesheets

Concept 4: Qt Stylesheets provide a powerful mechanism for customizing the appearance of widgets in a Qt application, similar to how CSS is used for styling web pages. Here's a brief overview of how they work:

- **CSS-like Syntax:** Qt Stylesheets use a syntax similar to Cascading Style Sheets (CSS) in web development. They allow you to define the appearance of widgets using style rules.
- **Selector and Declaration:** Each stylesheet rule consists of a selector and a declaration block. The selector specifies which widget or widgets the rule applies to, and the declaration block defines one or more properties to style these widgets.

Example:

```
button = new QPushButton("Example", this);
button->setGeometry(50,50,150,100);
button->setStyleSheet("QPushButton {"
                      "  border: 2px solid black;"
                      "  border-radius: 50px;"
                      "  background-color: lightgray;"
                      "  color: black;"
                      "}"
                      "QPushButton:hover {"
                      "  background-color: gray;"
                      "}"
                      "QPushButton:pressed {"
                      "  background-color: darkgray;"
                      "}");
```