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

```
#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();
```

Fonts

Concept 1: Fonts in Qt are handled through the QFont class, which provides extensive support for defining and manipulating font properties. The QFont class encapsulates the characteristics of fonts, such as family, point size, weight, style, and more. Here's a breakdown of the key aspects and functionalities of fonts in Qt:

16.1 Creating the font object

```
#include <QFont>

QFont font;
font.setFamily("Arial");      // Set font family
font.setPointSize(12);        // Set font point size
font.setWeight(QFont::Bold);  // Set font weight to bold
font.setItalic(true);         // Set font style to italic
```

16.2 Specifying Properties at Construction

```
QFont font("Times", 10, QFont::Bold, true);
```

16.3 Using the font object

```
QFont font;
font.setFamily("Arial");      // Set font family
font.setPointSize(12);        // Set font point size
font.setWeight(QFont::Bold);  // Set font weight to bold
font.setItalic(true);         // Set font style to italic

QLabel *label = new QLabel("Hello World", this);
label->setFont(font);
```

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.

17.1 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";
```

17.2 Converting to QString

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

17.3 Converting numeric types to QString

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

17.4 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
```

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 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 2: 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 3: 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;"
                      "}");
```

Other QWidget Children

1. **QLineEdit**: Allows single line text editing.
2. **QTextEdit**: A rich text editor widget which allows multi-line text editing.
3. **QComboBox**: A combined button and dropdown list.
4. **QCheckBox**: A checkbox with a text label.
5. **QRadioButton**: A radio button, typically used in groups to select one of many options.
6. **QSlider**: A slider widget for selecting a value from a range.
7. **QSpinBox**: Allows input of a single number through a spin box interface.
8. **QDoubleSpinBox**: Like QSpinBox, but for decimal numbers.
9. **QProgressBar**: Shows progress of a lengthy operation.
10. **QScrollBar**: A scrollbar widget.
11. **QDial**: A circular dial used to select a value.
12. **QTabWidget**: Allows tabbed navigation between different widgets.
13. **QListWidget**: Displays a list of items.
14. **QTreeWidget**: Displays items in a tree structure.
15. **QTableWidget**: Displays items in a table format.
16. **QGroupBox**: Groups related widgets together in a box.
17. **QStackedWidget**: Stacks widgets on top of one another; only one is visible at a time.
18. **QMainWindow**: The main window class, providing a framework for windowed applications.
19. **QDialog**: Used for creating dialog windows.

QWidget Methods

1. **Display Control:**
   - void show()
   - void hide()
   - void setVisible(bool)
   - void setEnabled(bool)
2. **Geometry and Positioning:**
   - void setGeometry(int x, int y, int width, int height)
   - void move(int x, int y)
   - void resize(int width, int height)
   - QRect geometry() const
   - QSize size() const
   - int width() const
   - int height() const
   - QPoint pos() const
3. **Layout Management:**
   - void setLayout(QLayout *layout)
   - QLayout *layout() const
4. **Style and Appearance:**
   - void setStyleSheet(const QString &styleSheet)
   - QString styleSheet() const
   - void setPalette(const QPalette &palette)
   - QPalette palette() const
5. **Event Handling:**
   - void installEventFilter(QObject *filterObj)
   - bool event(QEvent *event)
   - void update()
6. **Parent-Child Relationship:**
   - void setParent(QWidget *parent)
   - QWidget *parentWidget() const
7. **Window and Dialog Functions:**
   - void setWindowTitle(const QString &title)
   - QString windowTitle() const
   - void setWindowIcon(const QIcon &icon)
   - QIcon windowIcon() const
   - void setWindowFlags(Qt::WindowFlags type)

- Qt::WindowFlags windowFlags() const
- bool isWindow() const
- bool isModal() const

8. **Size Constraints:**
   - void setFixedSize(int w, int h)
   - void setFixedSize(const QSize &size)
   - void setMinimumSize(int minw, int minh)
   - void setMaximumSize(int maxw, int maxh)
9. **Focus Handling:**
   - void setFocus()
   - bool hasFocus() const
   - void setFocusPolicy(Qt::FocusPolicy policy)

Responsive Design

Concept 4: Responsive design in Qt refers to the practice of creating user interfaces (UIs) that adapt to various screen sizes and resolutions, ensuring a consistent and functional experience across different devices. Qt, being a versatile framework for both widget-based and QML-based UI development, provides several tools and techniques to achieve responsive design:

1. **Layout Managers:**
   - Qt's layout managers (QHBoxLayout, QVBoxLayout, QGridLayout, etc.) automatically adjust the size and position of widgets within a window or a parent widget.
   - They respond to window resize events and reorganize the contained widgets accordingly, maintaining their relative positions and sizes.
2. **Size Policies:**
   - Widgets in Qt have size policies (QSizePolicy) that determine how they grow or shrink in response to available space.
   - These policies can be set to make widgets more flexible or rigid in their size adjustments.
3. **Scalable Units:**
   - Using scalable units like points or ems for dimensions instead of fixed pixel sizes helps maintain the UI's appearance across different screen resolutions.
   - Qt Quick's GridUnit and dp (density-independent pixels) are examples of scalable units.

23.1 Layout Managers

```
1 #include <QLayout>
```

23.1.1 QHBoxLayout

```
1 #include <QHBoxLayout>
```

- **Purpose:** QHBoxLayout arranges child widgets in a horizontal line.
- **Usage:** It's typically used when you want to place widgets next to each other from left to right.
- **Spacing and Alignment:** It automatically manages the spacing between widgets and can align them in various ways (left, center, right).
- **Stretch Factors:** You can assign stretch factors to child widgets to control how much space each widget occupies relative to the others.
- **Example:** Placing a label and a line edit horizontally, where the label is the description and the line edit is the field for user input.

23.1.2 QVBoxLayout

```
1 #include <QVBoxLayout>
```

- **Purpose:** QVBoxLayout arranges child widgets in a vertical column.
- **Usage:** It's used when you want to stack widgets on top of each other from top to bottom.
- **Spacing and Alignment:** Like QHBoxLayout, it manages the spacing and supports various alignment options (top, center, bottom).
- **Stretch Factors:** You can also assign stretch factors to dictate the relative space each widget takes up.
- **Example:** Creating a form layout where each label and input field pair is stacked vertically.

23.1.3 QHBoxLayout and QVBoxLayout Methods

1. **Methods from QBoxLayout:**
   - void **addWidget**(QWidget *widget, int stretch = 0, Qt::Alignment alignment = 0)
     - Adds a widget to the end of the layout with optional stretch factor and alignment.
   - void **addLayout**(QLayout *layout, int stretch = 0)
     - Adds a nested layout to the end of this layout with optional stretch factor.
   - void **addStretch**(int stretch = 0)
     - Adds a stretchable space with a stretch factor to the layout.
   - void **insertWidget**(int index, QWidget *widget, int stretch = 0, Qt::Alignment alignment = 0)
     - Inserts a widget into the layout at the specified index.
   - void **insertLayout**(int index, QLayout *layout, int stretch = 0)
     - Inserts a nested layout at a specified index in this layout.
   - void **insertStretch**(int index, int stretch = 0)
     - Inserts a stretchable space at a specified index in the layout.
   - bool **setStretchFactor**(QWidget *widget, int stretch)
     - Sets the stretch factor for a widget in this layout.
   - bool **setStretchFactor**(QLayout *layout, int stretch)
     - Sets the stretch factor for a nested layout in this layout.
2. **Methods from QLayout:**
   - void **setAlignment**(QWidget *widget, Qt::Alignment alignment)
     - Sets the alignment for a widget in this layout.
   - void **setAlignment**(QLayout *layout, Qt::Alignment alignment)
     - Sets the alignment for a nested layout in this layout.

- void **setSizeConstraint**(QLayout::SizeConstraint constraint)
  - Sets the size constraint for this layout.
- QLayout::SizeConstraint **sizeConstraint**() const
  - Returns the size constraint of this layout.
- void **setSpacing**(int spacing)
  - Sets the spacing between widgets in this layout.
- int **spacing**() const
  - Returns the spacing between widgets in this layout.
- void **setContentsMargins**(int left, int top, int right, int bottom)
  - Sets the margins around the layout.
- void **getContentsMargins**(int *left, int *top, int *right, int *bottom) const
  - Retrieves the margins around the layout.
- QRect **geometry**() const
  - Returns the geometry of the layout.
- bool **isActive**() const
  - Returns true if the layout is enabled; otherwise returns false.
- QWidget* **parentWidget**() const
  - Returns the parent widget of this layout.

23.2 QVBoxLayout Example

Consider the code snippet

```
1
2  // Create a container for our vboxlayout and the vboxlayout
   ↪  itself
3  QWidget* layoutContainer = new QWidget(this);
4  QVBoxLayout* layout = new QVBoxLayout;
5
6  // Create some buttons
7  QPushButton* button1 = new QPushButton("1", this);
8  QPushButton* button2 = new QPushButton("1", this);
9
10 // Add the buttons to the layout
11 layout->addWidget(button1);
12 layout->addWidget(button2);
13
14 // Set the container as the container
15 layoutContainer->setLayout(layout);
16
```

23.2.1 Importance of Setting the Layout at the End

- **Layout Initialization:** It's crucial to add all widgets you want to be managed by the layout before setting the layout on the container. This ensures that when the layout is applied, it already contains all the widgets it needs to manage.
- **Efficient Redrawing:** Setting a layout on a widget can trigger a redraw of the widget and its children. If you set the layout before adding all the widgets, there could be unnecessary redraws and layout recalculations each time a new widget is added.
- **Avoiding Layout Conflicts:** Once a layout is set on a widget, adding the widget to another layout would cause issues, as a widget (or layout) can only belong to one layout at a time. Setting the layout at the end avoids such conflicts or double management.

The resizeEvent override

Concept 5: The resizeEvent is an important event in Qt that is triggered whenever a widget undergoes a resize operation. This event is part of the event handling mechanism in Qt, which is central to its graphical user interface (GUI) framework.

- **What is resizeEvent?**
  - The resizeEvent is a function that is called automatically by the Qt framework when the size of a widget changes. This includes when the widget is first shown (as it is sized to fit the contents or the specified dimensions), and when it is resized manually by the user (like adjusting the size of a window).
- **Declaration and Usage:**
  - The resizeEvent is a protected member function of the QWidget class. It can be overridden in a subclass to implement custom behavior when the widget is resized.
  - The function signature is: void resizeEvent(QResizeEvent *event);
- **Purpose:**
  - The primary purpose of overriding resizeEvent is to perform tasks that are necessary when the widget changes size. This could include adjusting the layout of child widgets, reallocating resources, or redrawing graphics.
  - For example, in a custom widget displaying a graph, you might need to recalculate the graph's dimensions and redraw it to fit the new size.

24.1 Example

```
1  void MainWindow::resizeEvent(QResizeEvent* event) override {
2      // Call base class implementation (important for proper
   ↪  functionality)
3      QWidget::resizeEvent(event);
4
5      // Recalculate the width of the left divider
6      leftDividerEndX = (this->width() / 5) - 15;
7
8      for (auto it = buttons.begin(); it != buttons.end(); ++it) {
9          (*it)->setFixedWidth(leftDividerEndX);
10     }
11
12     if (selectorHead) {
13         selectorHead->setFixedWidth(leftDividerEndX);
14     }
15 }
16
```

this isa paragra to my map of my decoment lets play vim tuton