Qt 5 Blueprints

Qt is a cross-platform application development framework that provides great visual interfaces for users and intuitive APIs for developers. The current version, Qt 5, provides enormous modules, from threading to interface animations, to ease your pain when developing cross-platform applications.

Starting with the basic concepts and fundamentals of Qt 5, Qt 5 Blueprints helps you get familiarized with the layouts and widgets of Qt as you build and debug a test application in Qt Creator. You will pick up the Qt design philosophy throughout the book, which will definitely help you to script your code in a more elegant manner.

Who this book is written for

If you are a programmer looking for a truly cross-platform GUI framework to help you save your time by side-stepping the incompatibility between different platforms and building applications using Qt 5 for multiple targets, then this book is most certainly intended for you. It is assumed that you have a basic programming experience of C++ and fundamental knowledge about Qt.

What you will learn from this book

- Create graphical user interface applications with a customized layout and widgets
- Explore how the Model-View-Controller architecture works and relevant classes in Qt
- Develop a configurable application that is able to save and restore its own settings
- Write both static and dynamic plugins for Qt Widgets and Qt Quick applications
- Convert your single-threaded application to a non-blocking application by moving part of the application to another thread
- Enable your applications to support other languages dynamically


Syemon Huang

7.50 x 9.25
235 mm x 191 mm

$ 49.99 US
£ 32.99 UK

Prices do not include local sales tax or VAT

Free Sample
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 2 'Building a Beautiful Cross-platform Clock'
- A synopsis of the book’s content
- More information on Qt 5 Blueprints

About the Author

**Symeon Huang** is an amateur developer who's currently doing his master's degree at Trinity College, Dublin. He has been contributing to open source projects for several years. He has worked in various areas, including the maintenance of Linux servers, desktop application development, and image recognition and analysis.

Symeon has always been passionate about cool technology and elegant programming techniques. He has been programming Qt and QML applications for 2 years and has also been developing pure C and C++ programs for many years. Most of the projects he's working on can be found on his GitHub and Gitorious pages.
Qt 5 Blueprints

Qt has been developed as a cross-platform framework and has been provided free to the public for years. It's mainly used to build GUI applications. It also provides thousands of APIs for easier development.

Qt 5, the latest major version of Qt, has once again proven to be the most popular cross-platform toolkit. With all these platform-independent classes and functions, you only need to code once, and then you can make it run everywhere.

In addition to the traditional and powerful C++, Qt Quick 2, which is more mature, can help web developers to develop dynamic and reliable applications, since QML is very similar to JavaScript.

What This Book Covers

Chapter 1, Creating Your First Qt Application, takes you through the fundamental concepts of Qt, such as signals and slots, and helps you create your first Qt and Qt Quick applications.

Chapter 2, Building a Beautiful Cross-platform Clock, teaches you how to read and write configurations and handle cross-platform development.

Chapter 3, Cooking an RSS Reader with Qt Quick, demonstrates how to develop a stylish RSS Reader in QML, which is a script language quite similar to JavaScript.

Chapter 4, Controlling Camera and Taking Photos, shows you how to access camera devices through the Qt APIs and make use of the status and menu bars.

Chapter 5, Extending Paint Applications with Plugins, teaches you how to make applications extendable and write plugins, by using the Paint application as an example.

Chapter 6, Getting Wired and Managing Downloads, shows you how to utilize Qt's network module using the progress bar, as well as learning about threaded programming in Qt.

Chapter 7, Parsing JSON and XML Documents to Use Online APIs, teaches you how to parse JSON and XML documents in both Qt/C++ and Qt Quick/QML, which is essential to obtain data from online APIs.
Chapter 8, *Enabling Your Qt Application to Support Other Languages*, demonstrates how to make internationalized applications, translate strings using Qt Linguist, and then load translation files dynamically.

Chapter 9, *Deploying Applications on Other Devices*, shows you how to package and make your applications redistributable on Windows, Linux, and Android.

Chapter 10, *Don’t Panic When You Encounter These Issues*, gives you some solutions and advice for common issues during Qt and Qt Quick application development and shows you how to debug Qt and Qt Quick applications.
In this chapter, you will learn that Qt is a great tool to build cross-platform applications. A Qt/C++ clock example is used as a demonstration here. The topics covered in this chapter, which are listed here, are essential for any real-world applications. These are as follows:

- Creating a basic digital clock
- Tweaking the digital clock
- Saving and restoring settings
- Building on Unix platforms

Creating a basic digital clock

It's time to create a new project, so we will create a Qt Widgets application named Fancy_Clock.

We won't utilize any Qt Quick knowledge in this chapter.
Now, change the window title to *Fancy Clock* or any other name that you like. Then, the main window UI needs to be tailored because the clock is displayed at the top of the desktop. The menu bar, status bar, and toolbar are all removed. After that, we need to drag an **LCD Number** widget into **centralWidget**. Next, change the layout of **MainWindow** to **LayOut Horizontally** in order to autoresize the subwidget. The last thing that needs to be done to the UI file is to change **frameShape** to **NoFrame** under the **QFrame** column in the property of **lcdNumber**. If you've done this right, you'll get a prototype of a digital clock, as shown here:

![Prototype of a digital clock](image)

In order to update the LCD number display repeatedly, we have to make use of the **QTimer** class to set up a timer that emits a signal repetitively. In addition to this, we need to create a slot to receive the signal and to update the LCD number display to the current time. Thus, the **QTime** class is also needed. This is how the header file of **MainWindowmainwindow.h** will look now:

```cpp
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>

namespace Ui {
 class MainWindow;
}

class MainWindow : public QMainWindow {

```

[28]
Q_OBJECT

public:
   explicit MainWindow(QWidget *parent = 0);
   ~MainWindow();

private:
   Ui::MainWindow *ui;

private slots:
   void updateTime();
};

#include <QTimer>
#include <QTime>
#include "mainwindow.h"
#include "ui_mainwindow.h"

MainWindow::MainWindow(QWidget *parent) :
   QMainWindow(parent),
   ui(new Ui::MainWindow)
{
   ui->setupUi(this);

   QTimer *timer = new QTimer(this);
   connect(timer, &QTimer::timeout, this, &MainWindow::updateTime);
   timer->start(1000);
   updateTime();
}

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

void MainWindow::updateTime()
{
   As you can see, the only modification made here is the declaration of a private updateTime slot. As usual, we're supposed to define this slot in mainwindow.cpp, whose content is pasted here. Note that we need to include QTimer and QTime.

#include <QTimer>
#include <QTime>
#include "mainwindow.h"
#include "ui_mainwindow.h"

MainWindow::MainWindow(QWidget *parent) :
   QMainWindow(parent),
   ui(new Ui::MainWindow)
{
   ui->setupUi(this);

   QTimer *timer = new QTimer(this);
   connect(timer, &QTimer::timeout, this, &MainWindow::updateTime);
   timer->start(1000);

   updateTime();
}

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

void MainWindow::updateTime()
{

Building a Beautiful Cross-platform Clock

```cpp
QTime currentTime = QTime::currentTime();
QString currentTimeText = currentTime.toString("hh:mm");
if (currentTime.second() % 2 == 0) {
    currentTimeText[2] = ' '; 
}
ui->lcdNumber->display(currentTimeText);
}
```

Inside the `updateTime` slot, the `QTime` class is used to deal with the time, that is, the clock. This class can provide accuracy of up to 1 millisecond, if the underlying operating system supports it. However, `QTime` has nothing to do with the time zone or daylight saving time. It is, at least, sufficient for our little clock. The `currentTime()` function is a static public function, which is used to create a `QTime` object that contains the system's local time.

As for the second line of the `updateTime` function, we used the `toString` function provided by `QTime` to convert the time to a string, and then saved it in `currentTimeText`. The arguments that are passed to `toString` are in the format of the time string. The full list of expressions can be obtained from Qt Reference Documentation. The colon in the middle of the clock should be flashing, just as in the case of a real digital clock. Hence, we used an if statement to control this. The colon will vanish when the second's value is even, and it will reappear when the second's value is odd. Here, inside the if block, we used the `[2]` operator to get a modifiable reference of the third character because this is the only way to do direct modifications to a character inside a string. Here, the counting of the `currentTimeText` string starts from 0. Meanwhile, the `at()` function of `QString` returns a constant character, which you have no right to change. At last, this function will let `lcdNumber` display the time string. Now, let's get back to the constructor of `MainWindow`. After the initialization of the UI, the first thing it does is to create a `QTimer` object. Why can't we use a local variable? The answer to that question is because the local variables will be destroyed after the construction of `MainWindow`. If the timer has gone, there's no way to trigger `updateTime` repetitively. We don't use a member variable because there is no need to perform the declaration work in the header file, since we won't use this timer elsewhere.

The `QTimer` class is used to create a repetitive and single-shot timer. It will emit the `timeout` signal at constant intervals after `start` is called. Here, we create one timer and connect the `timeout` signal to the `updateTime` slot so that `updateTime` is called every second.
There is another important aspect in Qt called **parent-child mechanism**. Although it's not as well-known as signals and slots, it plays a crucial role in the development of the Qt applications. Basically speaking, when we create an QObject child with a parent or explicitly set a parent by calling `setParent`, the parent will add this QObject child to its list of children. Then, when the parent is deleted, it'll go through its list of children and delete each child. In most cases, especially in the design of a UI, the parent-child relationship is set up implicitly. The parent widget or layout automatically becomes the parent object to its children widgets or layouts. In other cases, we have to explicitly set the parent for a QObject child so that the parent can take over its ownership and manage the release of its memory. Hence, we pass the QObject parent, which is this, a MainWindow class to the constructor of QTimer. This ensures that QTimer will be deleted after MainWindow is deleted. That's why we don't have to explicitly write the delete statements in the destructor.

At the end of the constructor, we need to call `updateTime` explicitly, which will allow the clock to display the current time. If we don't do this, the application will display a zero for a second until the `timeout` signal is emitted by `timer`. Now, run your application; it will be similar to the following screenshot:
**Tweaking the digital clock**

It's time to make this basic digital clock look more beautiful. Let's add something like a transparent background, which sits on top of the frameless window. Using a transparent background can deliver a fantastic visual effect. While the frameless window hides window decorations, including a border and the title bar, a desktop widget, such as a clock, should be frameless and displayed on top of the desktop.

To make our clock translucent, simply add the following line to the constructor of `MainWindow`:

```cpp
setAttribute(Qt::WA_TranslucentBackground);
```

The effect of the `WA_TranslucentBackground` attribute depends on the composition managers on the X11 platforms.

A widget may have lots of attributes, and this function is used to switch on or switch off a specified attribute. It's turned on by default. You need to pass a false Boolean as the second argument to disable an attribute. The full list of `Qt::WidgetAttribute` can be found in the Qt Reference Documentation.

Now, add the following line to the constructor as well, which will make the clock look frameless and make it stay on top of the desktop:

```cpp
setWindowFlags(Qt::WindowStaysOnTopHint | Qt::FramelessWindowHint);
```

Similarly, `Qt::WindowFlags` is used to define the type of widget. It controls the behavior of the widget, rather than of its properties. Thus, two hints are given: one is to stay on top and the other is to be frameless. If you want to preserve old flags while setting new ones, you need to add them to the combination.

```cpp
setWindowFlags(Qt::WindowStaysOnTopHint | Qt::FramelessWindowHint | windowFlags());
```

Here, the `windowFlags` function is used to retrieve the window flags. One thing you may be interested to know is that `setWindowFlags` will result in the invisibility of the widget after the `show` function. So, you can either call `setWindowFlags` before the `show` function of the window or widget or call `show` again after `setWindowFlags`. 

---

[30]
After the modification to the constructor, this is how the clock is expected to look:

There is a useful trick that you can use to hide the clock from the taskbar. Of course, a clock doesn't need to be displayed among the applications in a taskbar. You should never set a flag such as Qt::Tool or Qt::ToolTip alone to achieve this because this will cause the exit behavior of the application to be abnormal. This trick is even simpler; here is the code of main.cpp:

```cpp
#include "mainwindow.h"
#include <QApplication>

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

    QWidget wid;
    MainWindow w(&wid);
    w.show();

    return a.exec();
}
```
Building a Beautiful Cross-platform Clock

The preceding code makes our `MainWindow` object a child of `QWidget wid`. The child widgets won't display on the taskbar because there should be only one top parent widget. Meanwhile, our parent widget, `wid`, doesn't even show. It's tricky, but it's the only one that does the trick without breaking any other logic.

Well, a new problem has just surfaced. The clock is unable to move and the only way to close it is by stopping it through the Qt Creator's panel or through a keyboard shortcut. This is because we declared it as a frameless window, which led to an inability to control it via a window manager. Since there is no way to interact with it, it's impossible to close it by itself. Hence, the solution to this problem is to write our own functions to move and close the clock.

Closing this application may be more urgent. Let's see how to reimplement some functions to achieve this goal. First, we need to declare a new `showContextMenu` slot to display a context menu and reimplement `mouseReleaseEvent`. The following code shows the content of `mainwindow.h`:

```cpp
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>

namespace Ui {
    class MainWindow;
}

class MainWindow : public QMainWindow {
    Q_OBJECT

public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();

private:
    Ui::MainWindow *ui;

private slots:
    void updateTime();
    void showContextMenu(const QPoint &pos);

protected:
    void mouseReleaseEvent(QMouseEvent *);
};

#endif // MAINWINDOW_H
```
There are two new classes defined in the preceding code: QPoint and QMouseEvent. The QPoint class defines a point in the plane by using integer precision. Relatively, there is another class named QPointF, which provides float precision. Well, the QMouseEvent class inherits QEvent and QInputEvent. It contains some parameters that describe a mouse event. Let's see why we need them inmainwindow.cpp:

```cpp
#include <QTimer>
#include <QTime>
#include <QMouseEvent>
#include <QMenu>
#include <QAction>
#include "mainwindow.h"
#include "ui_mainwindow.h"

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    
    setAttribute(Qt::WA_TranslucentBackground);
    setWindowFlags(Qt::WindowStaysOnTopHint |
        Qt::FramelessWindowHint | windowFlags());
    
    connect(this, &MainWindow::customContextMenuRequested, this,
            &MainWindow::showContextMenu);
    
    QTimer *timer = new QTimer(this);
    connect(timer, &QTimer::timeout, this, &MainWindow::updateTime);
    timer->start(1000);
    
    updateTime();
}

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

void MainWindow::updateTime()
{
    QTime currentTime = QTime::currentTime();
    QString currentTimeText = currentTime.toString("hh:mm");
    if (currentTime.second() % 2 == 0) {
        
        
    }
}``
Building a Beautiful Cross-platform Clock

```cpp
currentTimeText[2] = ' ';
ui->lcdNumber->display(currentTimeText);
}

void MainWindow::showContextMenu(const QPoint &pos)
{
    QMenu contextMenu;
    contextMenu.addAction(QString("Exit"), this, SLOT(close()));
    contextMenu.exec(mapToGlobal(pos));
}

void MainWindow::mouseReleaseEvent(QMouseEvent *e)
{
    if (e->button() == Qt::RightButton) {
        emit customContextMenuRequested(e->pos());
    } else {
        QMainWindow::mouseReleaseEvent(e);
    }
}
```

Note that you should include `QMouseEvent`, `QMenu`, and `QAction` in order to utilize these classes. There is a predefined `customContextMenuRequested` signal, which is coupled with the newly created `showContextMenu` slot. For the sake of consistency, we will follow the rule that `Qt` defined, which means that the `QPoint` argument in `customContextMenuRequested` should be a local position instead of a global position. That's why we need a `mapToGlobal` function to translate `pos` to a global position. As for the `QMenu` class, it provides a menu widget for a menu bar, context menu, or other pop-up menus. So, we create the `contextMenu` object, and then add a new action with the `Exit` text. This is coupled with a `close` slot of `MainWindow`. The last statement is used to execute the `contextMenu` object at the specified global position. In other words, this slot will display a pop-up menu at the given position.

The reimplementation of `mouseReleaseEvent` is done to check the triggered button of the event. If it's the right button, emit the `customContextMenuRequested` signal with the local position of the mouse. Otherwise, simply call the default `mouseReleaseEvent` function of ` QMainWindow`.

Make use of the default member functions of the base class when you reimplement it.
Run the application again; you can quit by right-clicking on it and then selecting Exit. Now, we should continue the reimplementation to make the clock movable. This time, we need to rewrite two protected functions: `mousePressEvent` and `mouseMoveEvent`. Therefore, this is how the header file looks:

```cpp
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>

namespace Ui {
    class MainWindow;
}

class MainWindow : public QMainWindow {
    Q_OBJECT

public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();

private:
    Ui::MainWindow *ui;
    QPoint m(mousePos);

private slots:
    void updateTime();
    void showContextMenu(const QPoint &pos);

protected:
    void mousePressEvent(QMouseEvent *);
    void mouseMoveEvent(QMouseEvent *);
    void mousePressEvent(QMouseEvent *);
    void mouseMoveEvent(QMouseEvent *);
};

#endif // MAINWINDOW_H
```
Building a Beautiful Cross-platform Clock

There is also a declaration of a new private member variable in the preceding code, m_mousePos, which is a QPoint object used to store the local position of the mouse. The following code defines mousePressEvent and mouseMoveEvent:

```cpp
void MainWindow::mousePressEvent(QMouseEvent *e)
{
    m_mousePos = e->pos();
}

void MainWindow::mouseMoveEvent(QMouseEvent *e)
{
    this->move(e->globalPos() - m_mousePos);
}
```

It's easier than you thought. When a mouse button is pressed, the local position of the mouse is stored as m_mousePos. When the mouse is moving, we call the move function to move MainWindow to a new position. Because the position passed to move is a global position, we need to use globalPos of the event minus the local position of the mouse. Confused? The m_mousePos variable is the mouse's relative position to the top-left point of the parent widget, which is MainWindow in our case. The move function will move the top-left point of MainWindow to the given global position. While the e->globalPos() function is the global position of the mouse and not MainWindow, we need to subtract the relative position of m_mousePos to translate the mouse's global position to the top-left point position of MainWindow. After all this effort, the clock should look much more satisfying.

Saving and restoring settings

Although the clock can be moved, it won't restore its last position after restarting. In addition to this, we can give users some choices to adjust the clock's appearance, such as the font color. To make it work, we need the QSettings class, which provides platform-independent persistent settings. It needs a company or organization name and the name of an application. A typical QSettings object can be constructed by using this line:

```cpp
QSettings settings("Qt5 Blueprints", "Fancy Clock");
```

Here, Qt5 Blueprints is the organization's name and Fancy Clock is the application's name.
The settings are stored in the system registry on Windows, while they are stored in the XML preferences files on Mac OS X and the INI text files on the other Unix operating systems, such as Linux. However, we do not usually need to be concerned with this, since QSettings provides high-level interfaces to manipulate the settings.

If we're going to read and/or write settings in multiple places, we'd better set the organization and application in QCoreApplication, which is inherited by QApplication. The main.cpp file's content is shown as follows:

```cpp
#include "mainwindow.h"
#include <QApplication>

int main(int argc, char *argv[]) {
    QApplication a(argc, argv);
    a.setOrganizationName(QString("Qt5 Blueprints"));
    a.setApplicationName(QString("Fancy Clock"));
    QWidget wid;
    MainWindow w(&wid);
    w.show();
    return a.exec();
}
```

This enables us to use the default QSettings constructor to access the same settings.

In order to save the geometry and state of MainWindow, we need to reimplement closeEvent. First, we need to declare closeEvent to be a protected member function, as follows:

```cpp
void closeEvent(QCloseEvent *);
```

Then, let's define the closeEvent function inmainwindow.cpp, as follows:

```cpp
void MainWindow::closeEvent(QCloseEvent *e) {
    QSettings sts;
    sts.setValue("MainGeometry", saveGeometry());
    sts.setValue("MainState", saveState());
    e->accept();
}
```
Building a Beautiful Cross-platform Clock

Remember to add `#include <QSettings>` in order to include the QSettings header files.

Thanks to `setOrganizationName` and `setApplicationName`, we don't need to pass any arguments to the QSettings constructor now. Instead, we call a `setValue` function to save the settings. The `saveGeometry()` and `saveState()` functions return the `MainWindow` geometry and state respectively as the `QByteArray` objects.

The next step is to read these settings and restore the geometry and state. This can be done inside the constructor of `MainWindow`. You just need to add two statements to it:

```cpp
QSettings sts;
restoreGeometry(sts.value("MainGeometry").toByteArray());
restoreState(sts.value("MainState").toByteArray());
```

Here, `toByteArray()` can translate the stored value to a `QByteArray` object. How do we test to see if this works? To do this, perform the following steps:

1. Rebuild this application.
2. Run it.
4. Close it.
5. Run it again.

You'll see that the clock will appear at exactly the same position as it was before it closed. Now that you're pretty much familiar with widgets, layouts, settings, signals, and slots, it's time to cook a preference dialog by performing the following steps:

1. Right-click on the `Fancy_Clock` project in the `Projects` panel.
2. Select Add New....
3. Select Qt in the Files and Classes panel.
4. Click on Qt Designer Form Class in the middle panel.
5. Select Dialog with Buttons Bottom.
6. Fill in Preference under Class name.
7. Click on Next, and then select Finish.
Qt Creator will redirect you to the **Design** mode. First, let's change `windowTitle` to **Preference**, and then do some UI work. Perform the following steps to do this:

1. Drag **Label** to **QDialog** and change its `objectName` property to `colourLabel`. Next, change its text to **Colour**.
2. Add **QComboBox** and change its `objectName` property to `colourBox`.
3. Add the **Black**, **White**, **Green**, and **Red** items to `colourBox`.
4. Change the layout of **Preference** to **Lay Out in a Form Lay Out**.

Close this UI file. Go back to editing the `preference.h` add a private `onAccepted` slot. The following code shows the content of this file:

```cpp
#ifndef PREFERENCE_H
#define PREFERENCE_H

#include <QDialog>

namespace Ui { 
  class Preference;
}

class Preference : public QDialog 
{
  Q_OBJECT

public:
  explicit Preference(QWidget *parent = 0);
  ~Preference();

private:
  Ui::Preference *ui;

private slots:
  void onAccepted();
};

#endif // PREFERENCE_H
```
Building a Beautiful Cross-platform Clock

As usual, we define this slot in the source file. Besides, we have to set up some initializations in the constructor of Preference. Thus, preference.cpp becomes similar to the following code:

```cpp
#include <QSettings>
#include "preference.h"
#include "ui_preference.h"

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

    QSettings sts;
    ui->colourBox->setCurrentIndex(sts.value("Colour").toInt());

    connect(ui->buttonBox, &QDialogButtonBox::accepted, this,
            &Preference::onAccepted);
}

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

void Preference::onAccepted()
{
    QSettings sts;
    sts.setValue("Colour", ui->colourBox->currentIndex());
}
```
Similarly, we load the settings and change the current item of `colourBox`. Then, it's the signal and slot coupling that follow. Note that Qt Creator has automatically generated the accept and reject connections between `buttonBox` and `Preference` for us. The accepted signal of `buttonBox` is emitted when the OK button is clicked. Likewise, the rejected signal is emitted if the user clicks on Cancel. You may want to check Signals & Slots Editor in the Design mode to see which connections are defined there. This is shown in the following screenshot:

As for the definition of the onAccepted slot, it saves `currentIndex` of `colourBox` to the settings so that we can read this setting elsewhere.

Now, what we're going to do next is add an entry for Preference in the pop-up menu and change the color of `lcdNumber` according to the Colour setting value. Therefore, you should define a private slot and a private member function in `mainwindow.h` first.

```cpp
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>

namespace Ui {
    class MainWindow;
}

class MainWindow : public QMainWindow
{
    Q_OBJECT

private:
    // Private slots
    void onAccepted();

private:
    // Private member functions
    void saveColourToSettings();

public:
    MainWindow(QWidget *parent = nullptr);

    // Other public methods
};

#endif // MAINWINDOW_H
```

```cpp
#include <QMainWindow>

namespace Ui {
    class MainWindow;
}

class MainWindow : public QMainWindow
{
    Q_OBJECT

private:
    // Private slots
    void onAccepted();

private:
    // Private member functions
    void saveColourToSettings();

public:
    MainWindow(QWidget *parent = nullptr);

    // Other public methods
};
```
Building a Beautiful Cross-platform Clock

public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();

private:
    Ui::MainWindow *ui;
    QPoint m_mousePos;
    void setColour();

private slots:
    void updateTime();
    void showContextMenu(const QPoint &pos);
    void showPreference();

protected:
    void mouseReleaseEvent(QMouseEvent *);
    void mousePressEvent(QMouseEvent *);
    void mouseMoveEvent(QMouseEvent *);
    void closeEvent(QCloseEvent *); 

#ifdef MAINWINDOW_H

The setColour function is used to change the color of lcdNumber, while the showPreference slot will execute a Preference object. The definitions of these two members are in the mainwindow.cpp file, which is displayed in the following manner:

#include <QTimer>
#include <QTime>
#include <QMouseEvent>
#include <QMenu>
#include <QAction>
#include <QSettings>
#include "mainwindow.h"
#include "preference.h"
#include "ui_mainwindow.h"

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);
setAttribute(Qt::WA_TranslucentBackground);
setAttribute(Qt::WA_productName,
  "Qt::MainWindow::AttributeName", "MainWindow::AttributeName");
setAttribute(Qt::WA_TranslucentBackground);
setAttribute(Qt::WA_WindowIcon,
  "Qt::MainWindow::WindowIcon", MainWindow::WindowIcon);
void MainWindow::mouseReleaseEvent(QMouseEvent *e)
{
    if (e->button() == Qt::RightButton) {
        emit customContextMenuRequested(e->pos());
    } else {
        QMainWindow::mouseReleaseEvent(e);
    }
}

void MainWindow::mousePressEvent(QMouseEvent *e)
{
    m_mousePos = e->pos();
}

void MainWindow::mouseMoveEvent(QMouseEvent *e)
{
    this->move(e->globalPos() - m_mousePos);
}

void MainWindow::closeEvent(QCloseEvent *e)
{
    QSettings sts;
    sts.setValue("MainGeometry", saveGeometry());
    sts.setValue("MainState", saveState());
    e->accept();
}

void MainWindow::setColour()
{
    QSettings sts;
    int i = sts.value("Colour").toInt();
    QPalette c;
    switch (i) {
    case 0://black
        c.setColor(QPalette::Foreground, Qt::black);
        break;
    case 1://white
        c.setColor(QPalette::Foreground, Qt::white);
        break;
    case 2://green
        c.setColor(QPalette::Foreground, Qt::green);
        break;
    }
We call `setColour` in the constructor in order to set the color of `lcdNumber` correctly. Inside `setColour`, we first read the `Colour` value from the settings, and then use a `switch` statement to get the correct `QPalette` class before calling `setPalette` to change the color of `lcdNumber`. Since Qt doesn’t provide a direct way to change the foreground color of the `QLCDNumber` objects, we need to use this tedious method to achieve this. At the end of this member function, we call `update()` to update the `MainWindow` user interface.

![Don't forget to add the Preference action to contextMenu inside showContextMenu. Otherwise, there will be no way to open the dialog.](image)

In the relevant `showPreference` slot, we create a new `Preference` object, which is the child of `MainWindow`, and then call `exec()` to execute and show it. Lastly, we call `setColour()` to change the color of `lcdNumber`. As `Preference` is modal and `exec()` has its own event loop, it will block the application until `pre` is finished. After `pre` finishes executing, either by accepted or rejected, `setColour` will be called next. Of course, you can use the signal-slot way to implement it, but we have to apply some modifications to the previous code. Firstly, delete the `accepted-accept` signal-slot couple in `preference.ui` in the Design mode. Then, add `accept()` to the end of `onAccepted` in `preference.cpp`.

```cpp
void Preference::onAccepted()
{
    QSettings sts;
    sts.setValue("Colour", ui->colourBox->currentIndex());
    this->accept();
}
```
Building a Beautiful Cross-platform Clock

Now, showPreference inmainwindow.cpp can be rewritten as follows:

```cpp
void MainWindow::showPreference()
{
    Preference *pre = new Preference(this);
    connect(pre, &Preference::accepted, this,
            &MainWindow::setColour);
    pre->exec();
}
```

The `connect` statement shouldn't be placed after `exec()`, as it will cause the binding to fail.

No matter which way you prefer, the clock should have a Preference dialog now. Run it, select Preference from the pop-up menu, and change the color to whatever you please. You should expect a result similar to what is shown in the following screenshot:
Building on the Unix platforms

So far, we are still trapped with our applications on Windows. It's time to test whether our code can be built on other platforms. In this chapter, the code involved with only desktop operating systems, while we'll get a chance to build applications for mobile platforms later in this book. In terms of other desktop operating systems, there are plenty of them, and most of them are Unix-like. Qt officially supports Linux and Mac OS X, along with Windows. Hence, users of other systems, such as FreeBSD, may need to compile Qt from scratch or get prebuilt packages from their own communities. In this book, the Linux distribution Fedora 20 is used as a demonstration to introduce platform crossing. Please bear in mind that there are lots of desktop environments and theming tools on Linux, so don't be surprised if the user interface differs. Well, since you're curious, let me tell you that the desktop environment is KDE 4 with QtCurve, unifying GTK+ / Qt 4 / Qt 5 in my case. Let's get started as soon as you're ready. You can perform the following steps to do this:

1. Copy the source code of Fancy Clock to a directory under Linux.
2. Delete the Fancy_Clock.pro.user file.
3. Open this project in Qt Creator.

Now, build and run this application. Everything is good except that there's a taskbar icon. Small issues such as this can't be avoided without testing. Well, to fix this, just modify a single line in the constructor of MainWindow. Changing the window flags will amend this:

```cpp
setWindowFlags(Qt::WindowStaysOnTopHint | Qt::FramelessWindowHint | Qt::Tool);
```

If you run the file again, Fancy Clock won't show up in the taskbar any more. Please keep the MainWindow object, w, as a child of QWidget wid; otherwise, the application won't terminate after you click on Close.
Building a Beautiful Cross-platform Clock

Note that the Preference dialog uses native UI controls, rather than bringing the other platform's controls to this one. This is one of the most fascinating things that Qt has provided. All the Qt applications will look and behave like native applications across all platforms.

![Preference Dialog](image)

It's not a hustle but the truth is that once you code the Qt application, you can run it everywhere. You don't need to write different GUIs for different platforms. That dark age has long gone. However, you may want to write some functions for specific platforms, either because of particular needs or workarounds. Firstly, I'd like to introduce you to some Qt Add-On modules dedicated for several platforms.

Take Qt Windows Extras as an example. Some cool features that Windows provides, such as Thumbnail Toolbar and Aero Peek, are supported by Qt through this add-on module.

Well, adding this module to the project file directly, which in this case is Fancy_Clock.pro file, will definitely upset other platforms. A better way to do this is to test whether it's on Windows; if so, add this module to the project. Otherwise, skip this step. The following code shows you the Fancy_Clock.pro file, which will add the winextras module if it's built on Windows:

```plaintext
QT       += core gui

win32: QT += winextras

greaterThan(QT_MAJOR_VERSION, 4): QT += widgets

TARGET = Fancy_Clock
TEMPLATE = app

SOURCES += main.cpp\
    mainwindow.cpp \\
```

[48]
As you can see, `win32` is a conditional statement, which is `true` only if the host machine is Windows. After a `qmake` rerun for this project, you'll be able to include and utilize those extra classes.

Similarly, if you want to do something on the Unix platforms, simply use the keyword `unix`, but `unix` will be `true` only on Linux/X11 or Mac OS X. To distinguish Mac OS X from Linux, here's an example:

```cpp
win32 {
    message("Built on Windows")
} else: unix: macx{
    message("Built on Mac OS X")
} else {
    message("Built on Linux")
}
```

In fact, you can just use `unix: !macx` as the conditional statement to do some platform-specific work on Linux. It's a common practice to have many platform-specific statements in the project file(s), especially when your project needs to be linked with other libraries. You have to specify different paths for these libraries on different platforms, otherwise the compiler will complain about missing libraries or unknown symbols.

In addition to this, you may want to know how to write platform-specific code while keeping it from other platforms. Similar to C++, it's a predefined macro that is handled by various compilers. However, these compiler macro lists may differ from one compiler to another. So, it is better to use `Global Qt Declarations` instead. I'll use a the following short example to explain this further:

```cpp
void MainWindow::showContextMenu(const QPoint &pos)
{
    QMenu contextMenu;
    #ifdef Q_OS_WIN
    contextMenu.addAction(QString("Options"), this,
        SLOT(showPreference()));
    ```
Building a Beautiful Cross-platform Clock

```cpp
#if defined(Q_OS_LINUX)
    contextMenu.addAction(QString("Profile"), this, SLOT(showPreference()));
#else
    contextMenu.addAction(QString("Preference"), this, SLOT(showPreference()));
#endif
contextMenu.addAction(QString("Exit"), this, SLOT(close()));
contextMenu.exec(mapToGlobal(pos));
```

The preceding code shows you the new version of `showContextMenu`. The `Preference` menu entry will use different texts on different platforms, namely Windows, Linux, and Mac OS X. Change your `showContextMenu` function and run it again. You'll see `Options` on Windows, `Profile` on Linux, and `Preference` on Mac OS X. Below is a list concerning the platform-specific macros. You can get a full description, including other macros, functions, and types on the QtGlobal document.

<table>
<thead>
<tr>
<th>Macro</th>
<th>Correspond Platform</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q_OS_ANDROID</td>
<td>Android</td>
</tr>
<tr>
<td>Q_OS_FREEBSD</td>
<td>FreeBSD</td>
</tr>
<tr>
<td>Q_OS_LINUX</td>
<td>Linux</td>
</tr>
<tr>
<td>Q_OS_IOS</td>
<td>iOS</td>
</tr>
<tr>
<td>Q_OS_MAC</td>
<td>Mac OS X and iOS (Darwin-based)</td>
</tr>
<tr>
<td>Q_OS_WIN</td>
<td>All Windows platforms, including Windows CE</td>
</tr>
<tr>
<td>Q_OS_WINPHONE</td>
<td>Windows Phone 8</td>
</tr>
<tr>
<td>Q_OS_WINRT</td>
<td>Windows Runtime on Windows 8, Windows RT and Windows Phone 8</td>
</tr>
</tbody>
</table>

Summary

In this chapter, information, including some tricks, about UI designing is included. Furthermore, there are basic yet useful cross-platform topics. Now, you're able to write an elegant Qt application in your favorite, and possibly already mastered, C++.

In the next chapter, we are going to learn how to write an application in Qt Quick. However, fear not; Qt Quick is even easier and, of course, quicker to develop.
Where to buy this book

You can buy Qt 5 Blueprints from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.