Chapter No.10
"Localizing Your Application with Qt Linguist"
In this package, you will find:
The author’s biography
A preview chapter from the book, Chapter no.10 "Localizing Your Application with Qt Linguist"
A synopsis of the book’s content
Information on where to buy this book

About the Author

Ray Rischpater is an engineer and author, with over 20 years of experience in writing about mobile computing platforms and developing for them. During this time, he has participated in the development of Internet technologies and custom applications for Java ME, Qualcomm Brew™, Apple iPhone, Google Android, Palm OS, Newton, and Magic Cap, as well as several proprietary platforms. Presently, he’s employed as a Senior Engineer at Microsoft in Mountain View, and he works on mapping and data visualization.

When he is not writing for software development or about it, he enjoys hiking and photography with his family and friends in and around the San Lorenzo Valley in Central California. Whenever he's able to, he provides a public service through amateur radio as the licensed Amateur Extra station KF6GPE.


He holds a Bachelor's degree in Pure Mathematics from the University of California, Santa Cruz, and is a member of the IEEE, ACM, and ARRL.

For More Information:
I'd like to thank Ritika Singh for shepherding this book through the publishing process at Packt Publishing as well as for her suggestions throughout the process. I'd also like to thank the technical reviewers who provided their time to give suggestions and clarifications for each draft of the book. Of course, in the end, I'm responsible for any errors that might remain!
Application Development with
Qt Creator
Second Edition

Whether you're just getting started with programming or you've settled on Qt as the GUI toolkit for your project, Qt Creator is a great choice for an Integrated Development Environment (IDE). In this book, we work to help you make the most of Qt and Qt Creator, showing you almost every facet of using Qt Creator, from its configuration through compiling and debugging applications along with numerous tips and tricks. Along the way, you gain valuable experience not just with Qt Creator as an IDE, but with Qt and Qt Quick as well. After reading this book, you'll be able to:

- Edit, compile, debug, and run C++ applications using Qt Creator, opening a path to build state-of-the-art console and GUI applications with Qt and with the Standard Template Library (STL)
- Edit, compile, debug, and run Qt Quick applications using Qt Creator, giving you access to one of the most advanced declarative GUI authoring environments anywhere
- Design GUI applications using Qt Designer to build either traditional widget-based or Qt Quick applications
- Analyze the memory and the runtime performance of your Qt applications, make improvements, and fix defects
- Provide localized versions of your application so that you can deploy it all over the world in different languages
- Use Qt Quick and Qt Widgets to write mobile applications for platforms such as Google Android
- Build multimedia- and sensor-aware applications with Qt's support for multimedia and sensors

For More Information:
What This Book Covers

This book is divided into thirteen chapters, which you should plan to read in order, especially if you're new to Qt Creator and Qt programming in general. These chapters are given as follows:

Chapter 1, Getting Started with Qt Creator, shows you how to download and install Qt Creator as well as edit simple applications to test your installation.

Chapter 2, Building Applications with Qt Creator, shows you how to compile, run, and debug your application using Qt Creator. You will learn how Qt Creator integrates with both the GNU debugger and the Microsoft console debugger to provide breakpoints, memory inspection, and other debugging help.

Chapter 3, Designing Your Application with Qt Designer, shows you how to use the drag-and-drop GUI designer that is a part of Qt Creator to build both Qt Widget-based applications and Qt Quick applications.

Chapter 4, Qt Foundations, takes you through the foundations of software development using Qt and also covers its support for platform-agnostic application development.

Chapter 5, Developing Applications with Qt Widgets, shows you how to build applications using Qt Widgets that look and act like native desktop applications on the platform of your choice.

Chapter 6, Drawing with Qt, shows the various ways you can move beyond the built-in controls in Qt and make your own drawing on the screen and other drawable entities such as image files in PNG or JPEG.

Chapter 7, Doing More with Qt Quick, expands on what you learned about Qt Quick in the introductory chapters.

Chapter 8, Multimedia and Qt Quick, introduces you to Qt Quick's support for multimedia, such as audio and video playback as well as how to use a camera if it is connected.

Chapter 9, Sensors and Qt Quick, shows you how to use the various sensors on many of the devices available today using Qt Quick.

Chapter 10, Localizing Your Application with Qt Linguist, shows you how to manage resource strings for different locales, letting you build your application with different languages in different locales.

Chapter 11, Optimizing Performance with Qt Creator, shows you how to use Qt Creator to examine your Qt Quick application's runtime performance, as well as how to perform the memory profiling of your application with Valgrind, an open source diagnostic tool.

For More Information:

Chapter 12, Developing Mobile Applications with Qt Creator, gives you a glimpse of the exciting arena of mobile software development and shows you how you can use what you've learned in this book about Qt and Qt Creator to write applications for platforms such as Google Android.

Chapter 13, Qt Tips and Tricks, is packed with tricks for using Qt and Qt Creator that will help you use the Qt framework and the Qt Creator IDE efficiently.

For More Information:
Localizing Your Application with Qt Linguist

Localization is an important yet commonly neglected part of software development today. Most authors of applications, irrespective of whether those applications are commercial or open source, hope to capture a large number of users for their applications. Increasingly, this means supporting multiple languages in multiple locales, often needing support for multiple languages in one locale (think of it as French and English coexisting in Canada).

Since a long time, Qt has had a framework for making applications easy to localize with tools that help you to avoid hardcoding strings in your application and a GUI named Qt Linguist to help manage translation. In this chapter, we will take a look at Qt's strategy for localization, discussing the three tools (lupdate, lrelease, and Qt Linguist) that Qt provides and how to use them, along with what you need to do as you write your application to take advantage of Qt's localization framework.

In this chapter, we will take a look at the following topics:

- Understanding the task of localization
- Marking strings for localization
- Localizing your application with QLinguist
- Including localized strings in your application
- Localizing special parameters—currencies and dates with QLocale

For More Information:
Understanding the task of localization

Localizing your application has several phases that typically overlap throughout a project’s life cycle. These phases are:

1. As you write your application, you place strings to localize in your application in a specific way so that Qt can identify the strings as needing localization.
2. Periodically, you extract all the strings in your application and give them to translators in order to translate.
3. Translators provide translations for the strings in your application.
4. You compile translation files with the translated strings for each language you want to support.
5. The `tr` and `qsTr` functions for C++ and QML let you identify the strings in your application that require localization. Qt provides four tools to facilitate these phases.
6. The `lupdate` command generates a list of the strings that need localization in your application.
7. Translators use Qt Linguist to provide translations of the strings in your application.
8. The `lrelease` command takes the translated strings from Qt Creator and packages them in a format for your application to consume.

The following figure shows how these phases interact:

Software development is iterative, and localization is no exception. Small projects might prefer to do the localization just once, or perhaps twice, waiting until the application is nearly done before submitting the application strings for localization. Larger applications, or larger companies with a dedicated staff of translators, might prefer a more iterative approach, going through the localization cycle several times throughout application development. Qt supports both models.

For More Information:

Marking strings for localization

All the way back in *Chapter 1, Getting Started with Qt Creator*, I told you to always mark your strings for localization using the `tr` and `qsTr` functions: `tr` for C++ and `qsTr` for QML strings. Doing so has two key advantages:

- First, it enables Qt to find every string that needs localization
- Second, if you install a Qt translator object in your application and provide a translation file, the strings you wrap with these functions are automatically replaced by their localized equivalent

Let’s examine the use of `tr` in more detail. All Qt objects that include the `Q_OBJECT` macro in their declaration include the `tr` function. You’ve seen it with one argument, as follows:

```cpp
button = new QPushButton(tr("&Quit"), this);
```

The leading `&` in the string isn't for the `tr` function, but it is for the keyboard accelerators; you can prefix a letter with `&` to assign a keyboard accelerator and it gets the default system (a key combination with `Alt` for Windows, `Command` for Apple, and `Alt` for Linux). The `tr` function uses the string you pass as the string in the user interface if no translated version of the string appears in the application’s current translation table, or it uses the string in the current translation table if one exists.

The `tr` function can take a second argument, a disambiguation context that `tr` uses for the same string that might require different translations:

```cpp
tr("&Copy", "Menu");
```

This function can also handle strings with plurals, as follows:

```cpp
tr("%n item(s) replaced", ", count);
```

Depending on the value of `count` and the locale, a different string is returned. So, a native English translation could return "0 items replaced", "1 item replaced", "2 items replaced", and so on, while a French translation could return "0 item remplacé", "1 item remplacé", "2 items remplacés", and so on.

The `qsTr` function in QML works similarly but does not have the flexibility that the `tr` method has for disambiguation or handling plurals.
Localizing your application with QLinguist

Once you've marked your strings using `tr` or `qsTr`, you need to generate a table of those strings for Qt Linguist to localize. You can do this using the `lupdate` command, which takes your `.pro` file and walks your sources to look for strings to localize and creates an XML file of the strings you need to translate for Qt Linguist. You need to do this once for each language you want to support. When doing this, it's best to name the resulting files systematically; one way to do this is to use the name of the project file, followed by a dash, followed by the ISO-639-2 language code for the language.

A concrete example is in order. This chapter makes use of `QtLinguistExample`; we can run `lupdate` using a command such as this to create a list of strings that we'll translate to Esperanto (ISO-639-2 language code EPO):

```
% lupdate -pro .\QtLinguistExample.pro -ts .\QtLinguistExample-epo.ts
```

Don't forget that the `%` character is the command prompt, which might differ from system to system.

Here, the `-pro` file indicates the `.pro` file that contains the list of sources to scan for strings to translate, and the `-ts` argument indicates the name of the translation file to be written. You'll need `lupdate` in your path, of course. How you set your path will depend on whether you're working on Windows, Mac OS X, or Linux, and where you've installed Qt. Some installations of Qt might update your path automatically, while others might not do so. On my Windows machine, for example, I can find `lupdate` at `C:\qt\5.1.0\msvc2012_64\bin\lupdate.exe`.

The `.ts` file is an XML file with tags to indicate the strings to be translated, their context in your application's source code, and so forth. Qt Linguist will save the translations to its output file, which is named with a QM suffix as well, but don't worry: `lupdate` is smart enough to not overwrite the existing translations if you run it again after providing some translations.

Qt Linguist is a GUI application; on starting this application, you'll see a screen very similar to the next screenshot:
To begin, you need to open a `.ts` file you generated, by navigating to File | Open, and choosing a translation file. You'll be prompted for the destination language, and then you're given a list of the strings found. You—or your translators—only need to walk through each string and enter the corresponding string in the translated language. As you do so, you can see the context of the string in the source code in the right-most pane; the line of the source from which the string was captured is highlighted.

Qt Linguist lets you track which strings you've translated and also those which still need translation. The icon to the left-hand side of each of the strings can be one of the following:

- A black question mark, indicating that a string is yet to be translated
- A yellow question mark, indicating that the string doesn't pass all of Qt Linguist's validation tests, but you're ignoring the failures
- An exclamation point, indicating that the string you've provided doesn't pass Qt Linguist's validation tests

For More Information:
Localizing Your Application with Qt Linguist

- A yellow checkbox, indicating that you've provided a translation, but Qt Creator might have found a problem with it
- A green checkbox, indicating that the string has been translated and is ready to go

Qt Linguist provides some simple validation tests, such as ensuring that strings with printf-style arguments have the same number of arguments in each translation.

Qt Linguist also supports phrase books; you might be able to download a phrase book with common strings already localized to the language you're targeting.

At any point, you can generate a translation file for inclusion in your application by running lrelease. For example, to create one for our Esperanto strings, we'd use lrelease as follows:

```bash
% lrelease .\QtLinguistExample-epo.ts .\QtLinguistExample-epo.qm
```

This takes the incoming .ts file and generates a .qm file with the strings. The .qm files are highly compressed binary files used by Qt directly in the process of rendering the application.

Including localized strings in your application

In order to supply translated strings to the `tr` and `qsTr` functions in your application, your application needs to include a `QTranslator` object to read the .qm files and replace the strings provided to `tr` and `qsTr` with their translated counterparts.

We can do this in your main entry point function, as follows:

```cpp
QApplication a(argc, argv);
QTranslator translator;
bool result = translator.load("QtLinguistExample-epo.qm");
a.installTranslator(&translator);

// Other window setup stuff goes here

return a.exec();
```

This code allocates a `QTranslator` object and loads the indicated translation file into the translator before installing it into the `QApplication` object. In this example, we're hardcoding the language in order to localize to Esperanto.
Note that if you want to support the locale as picked by the system, you might choose to do it this way:

```cpp
QString locale = QLocale::system().name();
QTranslator translator;
translator.load(QString("QtLinguistExample-") + locale);
```

The `QLocale` class here is a class for managing the system's locale. Here, we use it to determine the system's locale, and then we attempt to load the localized string file for the system's current locale.

For this to work, the .qm files for the application need to be locatable by the application. They should be in the output directory; one way to do this during development is to turn off shadow builds in Qt Creator, under Build Settings in Project Pane. As you build your application's installer — a platform-specific task outside the scope of this book—you need to include your .qm files with the application binary.

For more information on Qt Linguist, see its manual at http://qt-project.org/doc/qt-5/qtlinguist-index.html.

### Localizing special parameters – currencies and dates with QLocale

A common thing you might need to do is localize currencies and dates. Qt makes this easy, although the solution isn't obvious until you've thought about it a bit.

First, you need to know about the `QString arg` method. It replaces an escaped number with the formatted version of its argument; if we write:

```cpp
QString s = QString("%1 %2").arg("a").arg("b");
```

Then, `s` contains the string "a b". Second, you need to know about the `toString` method of `QLocale` which formats its argument in a locale-specific way.

So, we could write:

```cpp
QString currencyValue = QString("%1 %2")
   .arg(tr("$")) .arg(QLocale::toString(value, 'g', 2))
```

This uses `tr` to localize the currency symbol and the `QLocale` class's static method, `toString`, to convert the value of the currency to a string with the locale-specific decimal separator (a period in the US and Canada, and a comma in Europe).
Date formatting is similar; the\texttt{toString} method of \texttt{QLocale} has overloads for the \texttt{QDateTime}, \texttt{QDate}, and \texttt{QTime} arguments, so you can simply write:

\begin{verbatim}
QDateTime whenDateTime = QDateTime::currentDateTime();
QString when = QLocale::toString(whenDateTime);
\end{verbatim}

This gets the current date and time and stores it in \texttt{whenDateTime} and then makes a string out of it using the locale's default formatting. The \texttt{toString} method can take a second argument that determines the output format; it's one of the following:

- \texttt{QLocale::LongFormat}: This uses the long version of month and day names
- \texttt{QLocale::ShortFormat}: This uses the short version of day and month names
- \texttt{QLocale::NarrowFormat}: This provides the narrowest form of formatting for the date and time

\section*{Summary}

Localizing applications with Qt is easy with Qt Linguist and the localization framework in Qt. To use the framework, though, you must mark your strings to localize with \texttt{tr} or \texttt{qsTr} in your source code wherever they appear. Once you do this, you can create a source file of strings to translate with QLinguist using Qt's \texttt{lupdate} command and then provide translations for each string. Once you've provided the translations, you compile them using \texttt{lrelease}, and then include them in your application by installing a \texttt{QTranslator} object in your application's main function and by loading the translation table generated by \texttt{lrelease}.

In the next chapter, we will take a look at another important aspect of software development that Qt Creator supports: performance analysis with the QML Profiler and Valgrind.
Where to buy this book


Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

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