Programming with Qt

Qt 3.0

Copyright © 2001 Trolltech AS. All rights reserved.

TROLLTECH, Qt and the Trolltech logo are registered trademarks of Trolltech AS. Linux is a registered trademark of Linus Torvalds. UNIX is a registered trademark of X/Open Company Ltd. Mac is a registered trademark of Apple Computer Inc. MS Windows is a registered trademark of Microsoft Corporation. All other products named are trademarks of their respective owners.

The definitive Qt documentation is provided in HTML format supplied with Qt, and available online at http://doc.trolltech.com. This PDF file was generated automatically from the HTML source as a convenience to users, although PDF is not an official Qt documentation format.

Contents

About Qt 4
How to Buy Qt 5
Key Features in Qt 3.0 6
Mailing Lists 12
Setting the Application Icon 13
Standard Accelerator Keys 14
Layout Classes 16
Keyboard Focus Overview 20
Drag and Drop 23
Qt Object Model 28
Object Trees and Object Ownership 29
Meta Object System 30
Properties 31
Using the Meta Object Compiler 34
Signals and Slots 41
Why doesn't Qt use templates for signals and slots? 46
Timers 49
Debugging Techniques 51
Class Inheritance Hierarchy 54
The Feature Definition File 64
Porting to Qt 3.x 66
Session Management 79
Shared Classes 82
Year 2000 Compliance Statement 86
Common Issues and Special Use Cases 87
How to Report A Bug 90
Installing Qt/Windows 91
Installing Qt/X11 92
Qt Free Edition License Agreement 94
Licenses for Code Used in Qt 96

GNU General Public License **99**
Index **106**

About Qt

Qt is a cross-platform C++ GUI application framework. It provides application developers with all the functionality needed to build state-of-the-art graphical user interfaces. Qt is fully object-oriented, easily extensible, and allows true component programming.

Since its commercial introduction in early 1996, Qt has formed the basis of many thousands of successful applications worldwide. Qt is also the basis of the popular KDE Linux desktop environment, a standard component of all major Linux distributions.

Qt is supported on the following platforms:

- **MS/Windows** - 95, 98, NT 4.0, ME, and 2000
- **Unix/X11** - Linux, Sun Solaris, HP-UX, Digital Unix, IBM AIX, SGI IRIX and a wide range of others
- **Macintosh** - Mac OS X
- **Embedded** - Linux platforms with framebuffer support.

Qt is a product of Trolltech.

Qt is released in different editions:

Qt Enterprise Edition and **Qt Professional Edition** provide for commercial software development. They permits traditional commercial software distribution and includes free upgrades and Technical Support Service. For the latest prices, please see the Trolltech web site, Pricing and Availability page, or contact sales@trolltech.com. The Enterprise Edition offers extended modules over the Professional Edition.

Qt Free Edition is the Unix/X11 version of Qt available for development of *Free and Open Source software* only. It is provided free of charge under the terms of both the Q Public License and the GNU General Public License. The latest version is available for download.

Qt/Embedded Free Edition is the Embedded version of Qt available for development of *Free software* only. It is provided free of charge under the terms of the GNU General Public License.

How to Buy Qt

You can order the Qt Enterprise Edition and Qt Professional Edition by printing out the order form and faxing it to +47 21 60 48 01. For prices, please see the Pricing and Availability page.

We accept payment by:

- Credit card (VISA, American Express, Mastercard, JCB)
- Cheque (US or international)
- SWIFT international bank transfer (see details on the order form)

(Due to restrictions imposed by the credit card companies, we cannot accept credit card orders via the web or e-mail; we must receive the signed order form.)

Cheques and order forms can also be sent by ordinary mail to:

Trolltech AS Waldemar Thranes gate 98 N-0175 Oslo NORWAY

For questions regarding purchase and pricing, please contact us at sales@trolltech.com or phone **+47 21 60 48 00**.

Key Features in Qt 3.0

Qt 3.0 adds a lot of new features and improvements over the Qt 2.x series. Some internals have undergone major redesign and new classes and methods have been added.

We have tried to keep the API of Qt 3.0 as compatible as possible with the Qt 2.x series. For most applications only minor changes will be needed to compile and run them successfully using Qt 3.0.

One of the major new features that has been added in the 3.0 release is a module allowing you to easily work with databases. The API is platform independent and database neutral. This module is seamlessly integrated into Qt Designer, greatly simplifying the process of building database applications and using data aware widgets.

Other major new features include a plugin architecture. You can use your own and third party plugins your own applications. The Unicode support of Qt 2.x has been greatly enhanced, it now includes full support for scripts written from right to left (e.g. Arabic and Hebrew) and also provides improved support for Asian languages.

Many new classes have been added to the Qt Library. Amongst them are classes that provide a docking architecture (QDockArea/QDockWindow), a powerful rich text editor (QTextEdit), a class to store and access application settings (QSettings) and a class to create and communicate with processes (QProcess).

Apart from the changes in the library itself a lot has been done to make the development of Qt applications with Qt 3.0 even easier than before. Two new applications have been added: Qt Linguist is a tool to help you translate your application into different languages; Qt Assistant is an easy to use help browser for the Qt documentation that supports bookmarks and can search by keyword.

Another change concerns the Qt build system, which has been reworked to make it a lot easier to port Qt to new platforms. You can use this platform independent build system for your own applications.

The Qt Library

A large number of new features has been added to Qt 3.0. The following list gives an overview of the most important new and changed aspects of the Qt library. A full list of every new method follows the overview.

Database support

One of the major new features in Qt 3.0 is the SQL module that provides cross-platform access to SQL databases, making database application programming with Qt seamless and portable. The API, built with standard SQL, is database-neutral and software development is independent of the underlying database.

A collection of tightly focused C++ classes are provided to give the programmer direct access to SQL databases. Developers can send raw SQL to the database server or have the Qt SQL classes generate SQL queries automatically. Drivers for Oracle, PostgreSQL, MySQL and ODBC are available and writing new drivers is straightforward.

Tying the results of SQL queries to GUI components is fully supported by Qt's SQL widgets. These classes include a tabular data widget (for spreadsheet-like data presentation with in-place editing), a form-based data browser (which provides data navigation and edit functions) and a form-based data viewer (which provides read-only forms). This framework can be extended by using custom field editors, allowing for example, a data table to use custom widgets for in-place editing. The SQL module fully supports Qt's signal/slots mechanism, making it easy

for developers to include their own data validation and auditing code.

Qt Designer fully supports Qt's SQL module. All SQL widgets can be laid out within Qt Designer, and relationships can be established between controls visually. Many interactions can be defined purely in terms of Qt's signals/slots mechanism directly in Qt Designer.

Plugins

The QLibrary class provides a platform independent wrapper for runtime loading of shared libraries. QPluginManager makes it trivial to implement plugin support in applications. The Qt library is able to load additional styles, database drivers and text codecs from plugins.

Qt Designer supports custom widgets in plugins, and will use the widgets both when designing and previewing forms.

Rich text engine and editor

The rich text engine originally introduced in Qt 2.0 has been further optimized and extended to support editing. It allows editing formatted text with different fonts, colors, paragraph styles, tables and images. The editor supports different word wrap modes, command-based undo/redo, multiple selections, drag and drop, and many other features. The engine is highly optimized for proccesing and displaying large documents quickly and efficiently.

Unicode

Apart from the rich text engine, another new feature of Qt 3.0 that relates to text handling is the greatly improved Unicode support. Qt 3.0 includes an implementation of the bidirectional algorithm (BiDi) as defined in the Unicode standard and a shaping engine for Arabic, which gives full native language support to Arabic and Hebrew speaking people. At the same time the support for Asian languages has been greatly enhanced.

The support is almost transparent for the developer using Qt to develop their applications. This means that developers who developed applications using Qt 2.x will automatically gain the full support for these languages when switching to Qt 3.0. Developers can rely on their application to work for people using writing systems different from Latin1, without having to worry about the complexities involved with these scripts, as Qt takes care of this automatically.

Docked and Floating Windows

Qt 3.0 introduces the concept of Dock Windows and Dock Areas. Dock windows are widgets, that can be attached to, and detached from, dock areas. The commonest kind of dock window is a tool bar. Any number of dock windows may be placed in a dock area. A main window can have dock areas, for example, QMainWindow provides four dock areas (top, left, bottom, right) by default. The user can freely move dock windows and place them at a convenient place in a dock area, or drag them out of the application and have them float freely as top level windows in their own right. Dock windows can also be minimized or hidden.

For developers, dock windows behave just like ordinary widgets. QToolbar for example is now a specialized subclass of a dock window. The API of QMainWindow and QToolBar is source compatible with Qt 2.x, so existing code which uses these classes will continue to work.

Regular Expressions

Qt has always provided regular expression support, but that support was pretty much limited to what was required in common GUI control elements such as file dialogs. Qt 3.0 introduces a new regular expression engine that supports most of Perl's regex features and is Unicode based. The most useful additions are support for parentheses (capturing and non-capturing) and backreferences.

Storing application settings

Most programs will need to store some settings between runs, for example, user selected fonts, colors and other preferences, or a list of recently used files. The new QSettings class provides a platform independent way to achieve this goal. The API makes it easy to store and retrieve most of the basic data types used in Qt (such as basic C++ types, strings, lists, colors, etc). The class uses the registry on the Windows platform and traditional resource files on Unix.

Creating and controlling other processes

QProcess is a class that allows you to start other programs from within a Qt application in a platform independent manner. It gives you full control over the started program, for example you can redirect the input and output of console applications.

Accessibility

Accessibility means making software usable and accessible to a wide range of users, including those with disabilities. In Qt 3.0, most widgets provide accessibility information for assistive tools that can be used by a wide range of disabled users. Qt standard widgets like buttons or range controls are fully supported. Support for complex widgets, like e.g. QListView, is in development. Existing applications that make use of standard widgets will become accessible just by using Qt 3.0.

Qt uses the Active Accessibility infrastructure on Windows, and needs the MSAA SDK, which is part of most platform SDKs. With improving standardization of accessibility on other platforms, Qt will support assistive technologies on other systems, too.

XML Improvements

The XML framework introduced in Qt 2.2 has been vastly improved. Qt 2.2 already supported level 1 of the Document Object Model (DOM), a W3C standard for accessing and modifying XML documents. Qt 3.0 has added support for DOM Level 2 and XML namespaces.

The XML parser has been extended to allow incremental parsing of XML documents. This allows you to start parsing the document directly after the first parts of the data have arrived, and to continue whenever new data is available. This is especially useful if the XML document is read from a slow source, e.g. over the network, as it allows the application to start working on the data at a very early stage.

SVG support

SVG is a W3C standard for "Scalable Vector Graphics". Qt 3.0's XML support means that QPicture can optionally generate and import static SVG documents. All the SVG features that have an equivalent in QPainter are supported.

Multihead support

Many professional applications, such as DTP and CAD software, are able to display data on two or more monitors. In Qt 3.0 the QDesktopWidget class provides the application with runtime information about the number and geometry of the desktops on the different monitors and such allows applications to efficiently use a multi-monitor setup.

The virtual desktop of Mac OS X, Windows 98, and 2000 is supported, as well as the traditional multi-screen and the newer Xinerama multihead setups on X11.

X11 specific enhancements

Qt 3.0 now complies with the NET WM Specification, recently adopted by KDE 2.0. This allows easy integration and proper execution with desktop environments that support the NET WM specification.

The font handling on X11 has undergone major changes. QFont no longer has a one-to-one relation with window system fonts. QFont is now a logical font that can load multiple window system fonts to simplify Unicode text display. This completely removes the burden of changing/setting fonts for a specific locale/language from the programmer. For end-users, any font can be used in any locale. For example, a user in Norway will be able to see Korean text without having to set their locale to Korean.

Qt 3.0 also supports the new render extension recently added to XFree86. This adds support for anti aliased text and pixmaps with alpha channel (semi transparency) on the systems that support the rendering extension (at the moment XFree 4.0.3 and later).

Printing

Printing support has been enhanced on all platforms. The QPrinter class now supports setting a virtual resolution for the painting process. This makes WYSIWYG printing trivial, and also allows you to take full advantage of the high resolution of a printer when painting on it.

The postscript driver built into Qt and used on Unix has been greatly enhanced. It supports the embedding of true/open type and type1 fonts into the document, and can correctly handle and display Unicode. Support for fonts built into the printer has been enhanced and Qt now knows about the most common printer fonts used for Asian languages.

QHttp

This class provides a simple interface for HTTP downloads and uploads.

Compatibility with the Standard Template Library (STL)

Support for the C++ Standard Template Library has been added to the Qt Template Library (QTL). The QTL classes now contain appropriate copy constructors and typedefs so that they can be freely mixed with other STL containers and algorithms. In addition, new member functions have been added to QTL template classes which correspond to STL-style naming conventions (e.g., push_back()).

Qt Designer

Qt Designer was a pure dialog editor in Qt 2.2 but has now been extended to provide the full functionality of a GUI design tool.

This includes the ability to lay out main windows with menus and toolbars. Actions can be edited within Qt Designer and then plugged into toolbars and menu bars via drag and drop. Splitters can now be used in a way similar to layouts to group widgets horizontally or vertically.

In Qt 2.2, many of the dialogs created by Qt Designer had to be subclassed to implement functionality beyond the predefined signal and slot connections. Whilst the subclassing approach is still fully supported, Qt Designer now offers an alternative: a plugin for editing slots. The editor offers features such as syntax highlighting, completion, parentheses matching and incremental search.

The functionality of Qt Designer can now be extended via plugins. Using Qt Designer's interface or by implementing one of the provided interfaces in a plugin, a two way communication between plugin and Qt Designer can be established. This functionality is used to implement plugins for custom widgets, so that they can be used as real widgets inside the designer.

Basic support for project management has been added. This allows you to read and edit *.pro files, add and remove files to/from the project and do some global operations on the project. You can now open the project file and have one-click access to all the *.ui forms in the project.

In addition to generating code via uic, Qt Designer now supports the dynamic creation of widgets directly from XML user interface description files (*.ui files) at runtime. This eliminates the need of recompiling your application when the GUI changes, and could be used to enable your customers to do their own customizations. Technically, the feature is provided by a new class, QWidgetFactory in the QResource library.

Qt Linguist

Qt Linguist is a GUI utility to support translating the user-visible text in applications written with Qt. It comes with two command-line tools: lupdate and lrelease.

Translation of a Qt application is a three-step process:

- Run lupdate to extract user-visible text from the C++ source code of the Qt application, resulting in a translation source file (a *.ts file).
- Provide translations for the source texts in the *.ts file using Qt Linguist.
- Run lrelease to obtain a light-weight message file (a *.qm file) from the *.ts file, which provides very fast lookup for released applications.

Qt Linguist is a tool suitable for use by translators. Each user-visible (source) text is characterized by the text itself, a context (usually the name of the C++ class containing the text), and an optional comment to help the translator. The C++ class name will usually be the name of the relevant dialog, and the comment will often contain instructions that describe how to navigate to the relevant dialog.

You can create phrase books for Qt Linguist to provide common translations to help ensure consistency and to speed up the translation process. Whenever a translator navigates to a new text to translate, Qt Linguist uses an intelligent algorithm to provide a list of possible translations: the list is composed of relevant text from any open phrase books and also from identical or similar text that has already been translated.

Once a translation is complete it can be marked as "done"; such translations are included in the *.qm file. Text that has not been "done" is included in the *.qm file in its original form. Although Qt Linguist is a GUI application with dock windows and mouse control, toolbars, etc., it has a full set of keyboard shortcuts to make translation as fast and efficient as possible.

When the Qt application that you're developing evolves (e.g. from version 1.0 to version 1.1), the utility lupdate merges the source texts from the new version with the previous translation source file, reusing existing translations. In some typical cases, lupdate may suggest translations. These translations are marked as unfinished, so you can easily find and check them.

Qt Assistant

Due to the positive feedback we received about the help system built into Qt Designer, we decided to offer this part as a separate application called Qt Assistant. Qt Assistant can be used to browse the Qt class documentation as well as the manuals for Qt Designer and Qt Linguist. It offers index searching, a contents overview, bookmarks history and incremental search. Qt Assistant is used by both Qt Designer and Qt Linguist for browsing their help documentation.

QMake

To ease portability we now provide the qmake utility to replace tmake. QMake is a C++ version of tmake which offers additional functionallity that is difficult to reproduce in tmake. Trolltech uses qmake in its build system for

Qt and related products and we have released it as free software.

Mailing Lists

Trolltech operates two mailing lists for Qt users: `qt-announce` and `qt-interest`.

qt-announce

The subscribers to this mailing list receives all official Trolltech announcements when new versions of Qt and other Trolltech products are released, and other Trolltech announcements of general interest.

To subscribe to the `qt-announce` mailing list, simply press the button below.

You can also subscribe by sending an email containing just the word `subscribe` or `subscribe` *`your@email.address`* to qt-announce-request@trolltech.com.

To unsubscribe yourself from the `qt-announce` mailing list, similarly send an email containing just the word `unsubscribe` or `unsubscribe` *`your@email.address`* to qt-announce-request@trolltech.com.

qt-interest

This mailing list is a discussion forum for Qt users. Typical traffic is 10 to 20 messages per day.

An archive of the messages from qt-interest is available on the Trolltech web site: Qt Mailing List Archive.

To subscribe to the `qt-interest` mailing list, send an email containing just the word `subscribe` or `subscribe` *`your@email.address`* to qt-interest-request@trolltech.com.

To unsubscribe yourself from the `qt-interest` mailing list, similarly send an email containing just the word `unsubscribe` or `unsubscribe` *`your@email.address`* to qt-interest-request@trolltech.com.

snapshot-users

There is a special mailing list, `snapshot-users@trolltech.com`, for discussion of Qt snapshot-related issues. To subscribe, send a message containing just the word "subscribe" (without the quotes) to snapshot-users-request@trolltech.com. We encourage you to use this mailing list in stead of `qt-interest` for snapshot-specific issues.

Trolltech does not sell or in any way redistribute the addresses on our mailing lists. We also employ active filtering to ensure the mailing lists are virtually free from spam.

Setting the Application Icon

The application icon, typically displayed in the upper left corner of the application top-level windows, is set by calling the QWidget::setIcon() method on top-level widgets.

In order to change the icon of the executable application file itself, as it is presented on the desktop (i.e. prior to application execution), it is necessary to employ another, platform-dependent technique.

Setting the Application Icon on Windows

First, create an ICO format bitmap file that contains the icon image. This can be done with e.g. Microsoft Visual C++: Select "File|New...", then select the "File" tab in the dialog that appears, and choose "Icon". (Note that you do not need to load your application into Visual C++; here we are only using its icon editor).

Store the ICO file in the source code directory of your application, for example, with the name, "myappico.ico". Then, create a text file called e.g. "myapp.rc" in which you put a single line of text:

```
IDI_ICON1               ICON    DISCARDABLE     "myappico.ico"
```

Lastly, assuming you are using qmake to generate your makefiles, add this line to your "myapp.pro" file:

```
RC_FILE = myapp.rc
```

Regenerate your makefile and your application. The .exe file will now be represented with your icon in e.g. Explorer.

If you do not use qmake, the necessary steps are: first, run the "rc" program on the .rc file, then link your application with the resulting .res file.

Standard Accelerator Keys

Applications invariably need to define accelerator keys for actions, and Qt provides functions to help with that, most importantly QAccel::shortcutKey().

Here is Microsoft's recommendations for accelerator key choice, with comments about the Open Group's recommendations where they exist and differ. For most commands, the Open Group either has no advice or agrees with Microsoft.

The boldfaced letter plus Alt is Microsoft's recommended choice, and we recommend supporting it. For an Apply button, for example, we recommend QButton::setText(tr("&Apply"));

If you have conflicting commands (e.g. About and Apply buttons in the same dialog), you're on your own.

- **A**bout
- Always on **T**op
- **A**pply
- **B**ack
- **B**rowse
- **C**lose (CDE: Alt+F4; Alt+F4 is "close window" in Windows)
- **C**opy (CDE: Ctrl+C, Ctrl+Insert)
- **C**opy Here
- Create **S**hortcut
- Create **S**hortcut Here
- Cu**t**
- **D**elete
- **E**dit
- **E**xit
- **E**xplore
- **F**ile
- **F**ind
- **H**elp
- Help **T**opics
- **H**ide
- **I**nsert
- Insert **O**bject
- **L**ink Here
- Ma**x**imize
- Mi**n**imize
- **M**ove

- **M**ove Here
- **N**ew
- **N**ext
- **N**o
- **O**pen
- Open **W**ith
- Page Set**u**p
- **P**aste
- Paste **L**ink
- Paste **S**hortcut
- Paste **S**pecial
- **P**ause
- **P**lay
- **P**rint
- **P**rint Here
- P**r**operties
- **Q**uick View
- **R**edo (CDE: Ctrl+Y, Alt+Backspace)
- **R**epeat
- **R**estore
- **R**esume
- **R**etry
- **R**un
- **S**ave
- Save **A**s
- Select **A**ll
- Se**n**d To
- **S**how
- **S**ize
- S**p**lit
- **S**top
- **U**ndo (CDE: Ctrl+Z or Alt+Backspace)
- **V**iew
- **W**hat's This?
- **W**indow
- **Y**es

There are also a lot of other keys and actions (that use other modifier keys than Alt). See the Microsoft and Open Group documentation for details.

The Microsoft book has ISBN 0735605661. The corresponding Open Group book is very hard to find, rather expensive and we cannot recommend it. However, if you really want it, OGPubs@opengroup.org might be able to help. Ask then for ISBN 1859121047.

Layout Classes

The Qt layout system provides a simple and powerful way of specifying the layout of child widgets.

By specifying the logical layout once, you get the following benefits:

- Positioning of child widgets.
- Sensible default sizes for top-level widgets.
- Sensible minimum sizes for top-level widgets.
- Resize handling.
- Automatic update when contents change:
  - Font size, text or other contents of subwidgets.
  - Hiding or showing a subwidget.
  - Removal of subwidget.

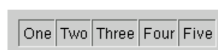
Qt's layout classes were designed for hand-written C++ code, so they're easy to understand and use.

The disadvantage of hand-written layout code is that it isn't very convenient when you're experimenting with the design of a form and you have to go through the compile, link and run cycle for each change. Our solution is Qt Designer, a GUI visual design tool which makes it fast and easy to experiment with layouts and which generates the C++ layout code for you.

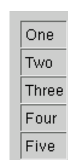
Layout Widgets

The easiest way to give your widgets a good layout is to use the layout widgets: QHBox, QVBox and QGrid. A layout widget automatically lays out its child widgets in the order they are constructed. To create more complex layouts, you can nest layout widgets inside each other.

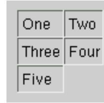
- A QHBox lays out its child widgets in a horizontal row, left to right.



- A QVBox lays out its child widgets in a vertical row, top to bottom.



- A QGrid lays out its child widgets in a two dimensional grid. You can specify how many columns the grid has, and it is populated left to right, beginning a new row when the previous row is full. The grid is fixed; the child widgets will not flow to other rows as the widget is resized.



The grid shown above can be produced by the following code:

```
QGrid *mainGrid = new QGrid( 2 ); // a 2 x n grid
new QLabel( "One", mainGrid );
new QLabel( "Two", mainGrid );
new QLabel( "Three", mainGrid );
new QLabel( "Four", mainGrid );
new QLabel( "Five", mainGrid );
```

You can adjust the layout somewhat by calling QWidget::setMinimumSize() or QWidget::setFixedSize() on the child widgets.

QLayout

When you add widgets to a layout the layout process works as follows:

1. All the widgets will initially be allocated an amount of space in accordance with their QWidget::sizePolicy().
2. If any of the widgets have stretch factors set, with a value greater than zero, then they are allocated space in proportion to their stretch factor.
3. If any of the widgets have stretch factors set to zero they will only get more space if no other widgets want the space. Of these, space is allocated to widgets with an Expanding size policy first.
4. Any widgets that are allocated less space than their minimum size (or minimum size hint in no minimum size is specified) are allocated this minimum size they require. (Widgets don't have to have a minimum size or minimum size hint in which case the strech factor is their determining factor.)
5. Any widgets that are allocated more space than their maximum size are allocated the maximum size space they require. (Widgets don't have to have a maximum size in which case the strech factor is their determining factor.)

Stretch Factors

Widgets are normally created without any stretch factor set. When they are laid out in a layout the widgets are given a share of space in accordance with their QWidget::sizePolicy() or their minimum size hint whichever is the greater. Stretch factors are used to change how much space widgets are given in proportion to one another.

If we have three widgets laid out using a QHBox with no stretch factors set we will get a layout like this:

If we apply stretch factors to each widget, they will be laid out in proportion (but never less than their minimum size hint), e.g.

QLayout

If you need more control over the layout, use a QLayout subclass. The layout classes included in Qt are QGridLayout and QBoxLayout. (QHBoxLayout and QVBoxLayout are trivial subclasses of QBoxLayout, that are simpler to use and make the code easier to read.)

When you use a layout, you have to insert each child both into its parent widget (done in the constructor) and into its layout (typically done with a function called addWidget()). This way, you can give layout parameters for each widget, specifying properties like alignment, stretch, and placement.

The following code makes a grid like the one above, with a couple of improvements:

```
QWidget *main = new QWidget;

// make a 1x1 grid; it will auto-expand
QGridLayout *grid = new QGridLayout( main, 1, 1 );

// add the first four widgets with (row, column) addressing
grid->addWidget( new QLabel( "One", main ), 0, 0 );
grid->addWidget( new QLabel( "Two", main ), 0, 1 );
grid->addWidget( new QLabel( "Three", main ), 1, 0 );
grid->addWidget( new QLabel( "Four", main ), 1, 1 );

// add the last widget on row 2, spanning from column 0 to
// column 1, and center aligned
grid->addMultiCellWidget( new QLabel( "Five", main ), 2, 2, 0, 1,
                          Qt::AlignCenter );

// let the ratio between the widths of columns 0 and 1 be 2:3
grid->setColStretch( 0, 2 );
grid->setColStretch( 1, 3 );
```

You can insert layouts inside a layout by giving the parent layout as a parameter in the constructor.

```
QWidget *main = new QWidget;
QLineEdit *field = new QLineEdit( main );
QPushButton *ok = new QPushButton( "OK", main );
QPushButton *cancel = new QPushButton( "Cancel", main );
QLabel *label = new QLabel( "Write once, compile everywhere.", main );

// a layout on a widget
QVBoxLayout *vbox = new QVBoxLayout( main );
vbox->addWidget( label );
vbox->addWidget( field );

// a layout inside a layout
QHBoxLayout *buttons = new QHBoxLayout( vbox );
buttons->addWidget( ok );
buttons->addWidget( cancel );
```

If you are not satisfied with the default placement, you can create the layout without a parent and then insert it with addLayout().

Custom Layouts

If the built-in layout classes are not sufficient, you can define your own. You must make a subclass of QLayout that handles resizing and size calculations, as well as a subclass of QGLayoutIterator to iterate over your layout class.

See the Custom Layout [Events, Actions, Layouts and Styles with Qt] page for an in-depth description.

Custom Widgets In Layouts

When you make your own widget class, you should also communicate its layout properties. If the widget has a QLayout, this is already taken care of. If the widget does not have any child widgets, or uses manual layout, you should reimplement the following QWidget member functions:

- QWidget::sizeHint() returns the preferred size of the widget.
- QWidget::minimumSizeHint() returns the smallest size the widget can have.
- QWidget::sizePolicy() returns a QSizePolicy; a value describing the space requirements of the widget.

Call QWidget::updateGeometry() whenever the size hint, minimum size hint or size policy changes. This will cause a layout recalculation. Multiple calls to updateGeometry() will only cause one recalculation.

If the preferred height of your widget depends on its actual width (e.g. a label with automatic word-breaking), set the hasHeightForWidth() flag in sizePolicy(), and reimplement QWidget::heightForWidth().

Even if you implement heightForWidth(), it is still necessary to provide a good sizeHint(). The sizeHint() provides the preferred width of the widget, and it is used by QLayout subclasses that do not support heightForWidth() (both QGridLayout and QBoxLayout support it).

For further guidance when implementing these functions, see their implementations in existing Qt classes that have similar layout requirements to your new widget.

Manual Layout

If you are making a one-of-a-kind special layout, you can also make a custom widget as described above. Reimplement QWidget::resizeEvent() to calculate the required distribution of sizes and call setGeometry() on each child.

The widget will get an event with type `LayoutHint` when the layout needs to be recalculated. Reimplement QWidget::event() to be notified of `LayoutHint` events.

Keyboard Focus Overview

Qt's widgets handle keyboard focus in the ways that have become customary in GUIs.

The basic issue is that the user's keystrokes can be directed at any of several windows on the screen, and any of several widgets inside the intended window. When the user presses a key, they expect it to go to the right place, and the software must try to meet this expectation. The system must determine which application the keystroke is directed at, which window within that application, and which widget within that window.

Focus motion

The customs which have evolved for directing keyboard focus to a particular widget are these:

1. The user presses Tab (or Shift+Tab) (or sometimes Enter).
2. The user clicks a widget.
3. The user presses a keyboard shortcut.
4. The user uses the mouse wheel.
5. The user moves the focus to a window, and the application has to figure out which widget within the window should get the focus.

Each of these motion mechanisms is different, and different types of widgets receive focus in only some of them. We'll cover each of them in turn.

Tab or Shift+Tab.

Pressing Tab is by far the most common way to move focus using the keyboard. Sometimes in data-entry applications Enter does the same as Tab. We will ignore that for the moment.

Pressing Tab, in all window systems in common use today, moves the keyboard focus to the next widget in a circular per-window list. Tab moves focus along the circular list in one direction, Shift+Tab in the other. The order in which Tab presses move from widget to widget is called the tab order.

In Qt, this list is kept in the QFocusData class. There is one QFocusData object per window, and widgets automatically append themselves to the end of it when QWidget::setFocusPolicy() is called with an appropriate QWidget::FocusPolicy. You can customize the tab order using QWidget::setTabOrder(). (If you don't, Tab generally moves focus in the order of widget construction.) Qt Designer provides a means of visually changing the tab order.

Since pressing Tab is so common, most widgets that can have focus should support tab focus. The major exception is widgets that are rarely used, and where there is some keyboard accelerator or error handler that moves the focus.

For example, in a data entry dialog, there might be a field that is only necessary in one per cent of all cases. In such a dialog, Tab could skip this field, and the dialog could use one of these mechanisms:

1. If the program can determine whether the field is needed, it can move focus there when the user finishes entry and presses OK, or when the user presses Enter after finishing the other fields. Alternately, include the

field in the tab order but disable it. Enable it if it becomes appropriate in view of what the user has set in the other fields.

2. The label for the field can include a keyboard shortcut that moves focus to this field.

There is also another exception to Tab support is text-entry widgets that must support tab; almost all text editors fall into this class. Qt treats Control+Tab as Tab and Control+Shift+Tab as Shift+Tab, and such widgets can reimplement QWidget::event() and handle Tab before calling QWidget::event() to get normal processing of all other keys. However, since some systems use Control+Tab for other purposes, and many users aren't aware of Control+Tab anyway, this isn't a complete solution.

The user clicks a widget.

This is perhaps even more common than pressing Tab on computers with a mouse or other pointing device.

Clicking to move the focus is slightly more powerful than Tab. While it moves the focus *to* a widget, for editor widgets it also moves the text cursor (the widget's internal focus) to the spot where the mouse is clicked.

Since it is so common, it's a good idea to support it for most widgets. People are used to it. However, there is also an important reason to avoid it: you may not want to remove focus from the widget where it was.

For example, in a word processor, when the user clicks the 'B' (boldface) tool button, what should happen to the keyboard focus? Should it remain where it was, almost certainly in the editing widget, or should it move to the 'B' button?

We advise supporting click-to-focus for widgets that support text entry, and to avoid it for most widgets where a mouse click has a different effect. (For buttons, we also recommend adding a keyboard shortcut - QButton and its subclasses make that very easy.)

In Qt, only the QWidget::setFocusPolicy() function affects click-to-focus.

The user presses a keyboard shortcut.

It's not unusual for keyboard shortcuts to move the focus. This can happen implicitly by opening modal dialogs, but also explicitly using focus accelerators such as those provided by QLabel::setBuddy(), QGroupBox and QTabBar.

We advise supporting shortcut focus for all widgets that the user may want to jump to. For example, a tab dialog can have keyboard shortcuts for each of its pages, so the user can press e.g. Alt+P to step to the Printing page. But don't overdo this - there are only a few keys, and it's also important to provide keyboard shortcuts for commands. Alt+P is also used for Paste, Play, Print and Print Here in the standard list of shortcuts, for example.

The user uses the mouse wheel.

On Microsoft Windows, mouse wheel usage is always handled by the widget that has keyboard focus. On Mac OS X and X11, it's handled by the widget that gets other mouse events.

The way Qt handles this platform difference is by letting widgets move the keyboard focus when the wheel is used. With the right focus policy on each widget, applications can work idiomatically correctly on Windows, Mac OS X, and X11.

The user moves the focus to this window.

In this situation the application has to determine which widget within the window should receive the focus.

This can be simple: if the focus has been in this window before, then the last widget to have focus should regain it. Qt does this automatically.

If focus has never been in this window before and you know where focus should start out, call QWidget::setFocus() on the widget which should receive focus before you QWidget::show() it. If you don't, Qt will pick some suitable widget.

Drag and Drop

Drag and drop provides a simple visual mechanism which users can use to transfer information between and within applications. (In the literature this is referred to as a "direct manipulation model".) Drag and drop is similar in function to the clipboard's cut-and-paste mechanism.

- Dragging
- Dropping
- The Clipboard
- Drag and Drop Actions
- Adding New Drag and Drop Types
- Advanced Drag-and-Drop
- Inter-operating with Other Applications

For drag and drop examples see (in increasing order of sophistication): `qt/examples/iconview/simple_dd`, `qt/examples/dragdrop` and `qt/examples/fileiconview`. See also the QMultiLineEdit widget source code.

Dragging

To start a drag, for example in a mouse motion event, create an object of the QDragObject subclass appropriate for your media, such as QTextDrag for text and QImageDrag for images. Then call the drag() method. This is all you need for simple dragging of existing types.

For example, to start dragging some text from a widget:

```
void MyWidget::startDrag()
{
  QDragObject *d = new QTextDrag( myHighlightedText(), this );
  d->dragCopy();
  // do NOT delete d.
}
```

Note that the QDragObject is not deleted after the drag. The QDragObject needs to persist after the drag is apparently finished - it may still be communicating with another process. Eventually Qt will delete the object. If the widget owning the drag object is deleted before then, any pending drop will be cancelled and the drag object deleted. For this reason, you should be careful what the object references.

Dropping

To be able to receive media dropped on a widget, call setAcceptDrops(TRUE) for the widget (e.g. in its constructor), and override the event handler methods dragEnterEvent() and dropEvent(). For more sophisticated applications overriding dragMoveEvent() and dragLeaveEvent() will also be necessary.

For example, to accept text and image drops:

```
MyWidget::MyWidget(...) :
  QWidget(...)
{
  ...
  setAcceptDrops(TRUE);
}

void MyWidget::dragEnterEvent(QDragEnterEvent* event)
{
  event->accept(
      QTextDrag::canDecode(event) ||
      QImageDrag::canDecode(event)
  );
}

void MyWidget::dropEvent(QDropEvent* event)
{
  QImage image;
  QString text;

  if ( QImageDrag::decode(event, image) ) {
    insertImageAt(image, event->pos());
  } else if ( QTextDrag::decode(event, text) ) {
    insertTextAt(text, event->pos());
  }
}
```

The Clipboard

The QDragObject, QDragEnterEvent, QDragMoveEvent, and QDropEvent classes are all subclasses of QMimeSource - the class of objects which provide typed information. If you base your data transfers on QDragObject, you not only get drag-and-drop, but you also get traditional cut-and-paste for free - the QClipboard has two functions:

```
setData(QMimeSource*)
QMimeSource* data()const
```

With these functions you can trivially put your drag-and-drop oriented information on the clipboard:

```
void MyWidget::copy()
{
  QApplication::clipboard()->setData(
      new QTextDrag(myHighlightedText())
  );
}

void MyWidget::paste()
{
  QString text;
  if ( QTextDrag::decode(QApplication::clipboard()->data(), text) )
      insertText( text );
}
```

You can even use QDragObject subclasses as part of file IO. For example, if your application has a subclass of QDragObject that encodes CAD designs in DXF format, your saving and loading code might be:

```
void MyWidget::save()
```

```
{
  QFile out(current_file_name);
  out.open(IO_WriteOnly);
  MyCadDrag tmp(current_design);
  out.writeBlock( tmp->encodedData( "image/x-dxf" ) );
}

void MyWidget::load()
{
  QFile in(current_file_name);
  in.open(IO_ReadOnly);
  if ( !MyCadDrag::decode(in.readAll(), current_design) ) {
      QMessageBox::warning( this, "Format error",
          tr("The file \"%1\" is not in any supported format")
           .arg(current_file_name)
      );
  }
}
```

Note how the QDragObject subclass is called "MyCadDrag", not "MyDxfDrag" - because in the future you might extend it to provide DXF, DWG, SVF, WMF, or even QPicture data to other applications.

Drag and Drop Actions

In the simpler cases, the target of a drag-and-drop receives a copy of the data being dragged and the source decides whether to delete the original. This is the "Copy" action in QDropEvent. The target may also choose to understand other actions, specifically the Move and Link actions. If the target understands the Move action, *the target* is responsible for both the copy and delete operations and the source will not attempt to delete the data itself. If the target understands the Link, it stores its own reference to the original information, and again the source does not delete the original. The most common use of drag-and-drop actions is when performing a Move within the same widget - see the Advanced Drag-and-Drop section below.

The other major use of drag actions is when using a reference type such as text/uri-list, where the dragged data are actually references to files or objects.

Adding New Drag and Drop Types

As suggested in the DXF example above, drag-and-drop is not limited to text and images. Any information can be dragged and dropped. To drag information between applications, the two applications must be able to indicate to each other which data formats they can accept and which they can produce. This is achieved using *MIME types* - the drag source provides a list of MIME types that it can produce (ordered from most appropriate to least appropriate), and the drop target chooses which of those it can accept. For example, QTextDrag provides support for the "`text/plain`" MIME type (ordinary unformatted text), and the Unicode formats "`text/utf16`" and "`text/utf8`"; QImageDrag provides for "`image/*`", where `*` is any image format that QImageIO supports; and the QUriDrag subclass provides "`text/uri-list`", a standard format for transferring a list of filenames (or URLs).

To implement drag-and-drop of some type of information for which there is no available QDragObject subclass, the first and most important step is to look for existing formats that are appropriate - the Internet Assigned Numbers Authority (IANA) provides a

hierarchical list of MIME media types at the Information Sciences Institute (ISI). Using standard MIME types maximizes the inter-operability of your application with other software now and in the future.

To support an additional media type, subclass either QDragObject or QStoredDrag. Subclass QDragObject when you need to provide support for multiple media types. Subclass the simpler QStoredDrag when one type is sufficient.

Subclasses of QDragObject will override the const char* format(int i) const and QByteArray encodedData(const char* mimetype) const members, and provide a set-method to encode the media data and static members canDe-

code() and decode() to decode incoming data, similar to bool canDecode(QMimeSource*) const and QByteArray decode(QMimeSource*) const of QImageDrag. Of course, you can provide drag-only or drop-only support for a media type by omitting some of these methods.

Subclasses of QStoredDrag provide a set-method to encode the media data and the same static members canDecode() and decode() to decode incoming data.

Advanced Drag-and-Drop

In the clipboard model, the user can *cut* or *copy* the source information, then later paste it. Similarly in the drag-and-drop model, the user can drag a *copy* of the information or they can drag the information itself to a new place (*moving* it). The drag-and-drop model however has an additional complication for the programmer: the program doesn't know whether the user wants to cut or copy until the drop (paste) is done! For dragging between applications, it makes no difference, but for dragging within an application, the application must take a little extra care not to tread on its own feet. For example, to drag text around in a document, the drag start point and the drop event might look like this:

```
void MyEditor::startDrag()
{
  QDragObject *d = new QTextDrag(myHighlightedText(), this);
  if ( d->drag() && d->target() != this )
    cutMyHighlightedText();
}

void MyEditor::dropEvent(QDropEvent* event)
{
  QString text;

  if ( QTextDrag::decode(event, text) ) {
    if ( event->source() == this && event->action() == QDropEvent::Move ) {
      // Careful not to tread on my own feet
      event->acceptAction();
      moveMyHighlightedTextTo(event->pos());
    } else {
      pasteTextAt(text, event->pos());
    }
  }
}
```

Some widgets are more specific than just a "yes" or "no" response when data is dragged onto them. For example, a CAD program might only accept drops of text onto text objects in the view. In these cases, the dragMoveEvent() is used and an *area* is given for which the drag is accepted or ignored:

```
void MyWidget::dragMoveEvent(QDragMoveEvent* event)
{
  if ( QTextDrag::canDecode(event) ) {
    MyCadItem* item = findMyItemAt(event->pos());
    if ( item )
      event->accept();
  }
}
```

If the computations to find objects are particularly slow, you might achieve improved performance if you tell the system an area for which you promise the acceptance persists:

```
void MyWidget::dragMoveEvent(QDragMoveEvent* event)
```

```
{
  if ( QTextDrag::canDecode(event) ) {
    MyCadItem* item = findMyItemAt(event->pos());
    if ( item ) {
      QRect r = item->areaRelativeToMeClippedByAnythingInTheWay();
      if ( item->type() == MyTextType )
        event->accept( r );
      else
        event->ignore( r );
    }
  }
}
```

The dragMoveEvent() can also be used if you need to give visual feedback as the drag progresses, to start timers, to scroll the window, or whatever is appropriate (don't forget to stop the scrolling and timers in a dragLeaveEvent() though).

Inter-operating with Other Applications

On X11, the public XDND protocol is used, while on Windows Qt uses the OLE standard, and Qt/Mac uses the Carbon Drag Manager. On X11, XDND uses MIME, so no translation is necessary. The Qt API is the same regardless of the platform. On Windows, MIME-aware applications can communicate by using clipboard format names that are MIME types. Already some Windows applications use MIME naming conventions for their clipboard formats. Internally, Qt has facilities for translating proprietary clipboard formats to and from MIME types. This interface will be made public at some time, but if you need to do such translations now, contact your Qt Technical Support service.

On X11, Qt also supports drops via the Motif Drag&Drop Protocol. The implementation incorporates some code that was originally written by Daniel Dardailler, and adapted for Qt by Matt Koss <koss@napri.sk> and Trolltech. Here is the original copyright notice:

Copyright 1996 Daniel Dardailler.

Permission to use, copy, modify, distribute, and sell this software for any purpose is hereby granted without fee, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Daniel Dardailler not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. Daniel Dardailler makes no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

Modifications Copyright 1999 Matt Koss, under the same license as above.

Qt Object Model

The standard C++ Object Model provides very efficient runtime support for the object paradigm. But the C++ Object Model's static nature is inflexibile in certain problem domains. Graphical User Interface programming is a domain that requires both runtime efficiency and a high level of flexibility. Qt provides this, by combining the speed of C++ with the flexibility of the Qt Object Model.

Qt adds these features to C++:

- a very powerful mechanism for seamless object communication called signals and slots,
- queryable and designable object properties,
- powerful events and event filters,
- contextual string translation for internationalization,
- sophisticated interval driven timers that make it possible to elegantly integrate many tasks in an event-driven GUI.
- hierarchical and queryable object trees that organize object ownership in a natural way.
- guarded pointers, QGuardedPtr, that are automatically set to null when the referenced object is destroyed, unlike normal C++ pointers which become "dangling pointers" when their objects are destroyed.

Many of these Qt features are implemented with standard C++ techniques, based on inheritance from QObject. Others, like the object communication mechanism and the dynamic property system, require the Meta Object System provided by Qt's own Meta Object Compiler (moc).

The Meta Object System is a C++ extension that makes the language better suited to true component GUI programming. Although templates can be used to extend C++, the Meta Object System provides benefits using standard C++ that cannot be achieved with templates; see Why doesn't Qt use templates for signals and slots?.

Object Trees and Object Ownership

QObjects organize themselves in object trees. When you create a QObject with another object as parent, it's added to the parent's children() list, and is deleted when the parent is. This turns out that this approach fits the needs of GUI objects very well. For example, a QAccel (keyboard accelerator) is a child of the relevant window, so when the user closes that window, the accelerator is deleted too.

The static function QObject::objectTrees() provides access to all the root objects that currently exist.

QWidget, the base class of everything that appears on the screen, extends the parent-child relationship. A child normally also becomes a child widget, i.e. it is displayed in its parent's coordinate system and is graphically clipped by its parent's boundaries. For example, when the an application deletes a message box after it has been closed, the message box's buttons and label are also deleted, just as we'd want, because the buttons and label are children of the message box.

You can also delete child objects yourself, and they will remove themselves from their parents. For example, when the user removes a toolbar it may lead to the application deleting one of its QToolBar objects, in which case the tool bar's QMainWindow parent would detect the change and reconfigure its screen space accordingly.

The debugging functions QObject::dumpObjectTree() and QObject::dumpObjectInfo() are often useful when an application looks or acts strangely.

Meta Object System

The Meta Object System in Qt is responsible for the signal/slot mechanism for communication between objects, runtime type information and the dynamic property system.

It is based on three things:

- the QObject class,
- the Q_OBJECT macro inside the private section of the class declaration,
- the Meta Object Compiler (moc).

The *moc* reads a C++ source file. If it finds one or more class declarations that contain the Q_OBJECT macro, it produces another C++ source file which contains the meta object code for the classes that contain the Q_OBJECT macro. This generated source file is either #included into the class' source file or compiled and linked with the class' implementation.

In addition to providing the signals and slots mechanism for communication between objects (the main reason for introducing the system), the meta object code implements certain other features in QObject:

- a function className() that returns the class name as a string at runtime, without requiring native runtime type information (RTTI) support through the C++ compiler.
- a function inherits() that returns whether an object is an instance of a class that inherits a specified class within the QObject inheritance tree.
- two functions tr() and trUtf8() for string translation as used for internationalization.
- two functions setProperty() and property() to dynamically set and get object properties by name.
- a function metaObject() that returns the associated meta object for the class.

While it is possible to use QObject as a base class without the Q_OBJECT macro and without meta object code, neither signals and slots nor the other features described here will be available if the Q_OBJECT macro is not used. From the meta object system's point of view, a QObject subclass without meta code is equivalent to its closest ancestor with meta object code. This means for example, that className() will not return the actual name of your class, but the class name of this ancestor. We strongly recommend that all subclasses of QObject use the Q_OBJECT macro regardless whether they actually use signals, slots and properties or not.

Properties

Qt provides a sophisticated property system similar to those shipped by some compiler vendors. However, as a compiler- and platform-independent library, Qt cannot rely on non-standard compiler features like __property or [property]. Our solution works with any standard C++ compiler on every platform we support. It's based on the meta-object system that also provides object communication through signals and slots.

The Q_PROPERTY macro in a class declaration declares a property. Properties can only be declared in classes that inherit QObject. A second macro, Q_OVERRIDE, can be used to override some aspects of an inherited property in a subclass.

To the outer world, a property appears quite similar to a data member. However, properties have several features that distinguish them from ordinary data members:

- A read function. This always exists.
- A write function. Optional: read-only properties like QWidget::isDesktop() do not have this.
- An attribute "stored" that indicates persistence. Most properties are stored, but a few virtual properties like QWidget::minimumWidth() aren't, since it's just way of looking at QWidget::minimumSize(), and has no data of its own.
- A reset function to set a property back to its context specific default value. Very unusual, but e.g. QWidget::font() needs this, since no call to QWidget::setFont() can mean 'reset to the context specific font'.
- An attribute "designable" that indicates whether it makes sense to set make the property available in a GUI builder (e.g. Qt Designer). For most properties this make sense, but not for all, e.g. QButton::isDown(). The user can press buttons, and the application programmer can make the program press its own buttons, but a GUI design tool can't press buttons.

The read, write and reset functions can be just about any member functions, inherited or not, virtual or not, with the exception of member functions inherited from parents other than the first in the case of multiple inheritance.

Properties can be read and written through generic functions in QObject without knowing anything about the class in use. These two function calls are equivalent:

```
// QButton *b and QObject *o point to the same button
b->setDown( TRUE );
o->setProperty( "down", TRUE );
```

Equivalent, that is, except that the first is faster, and provides much better diagnostics at compile time. When practical, the first is better. However, since you can get a list of all available properties for any QObject through its QMetaObject, QObject::setProperty() can give you control over classes that weren't available at compile time.

As well as QObject::setProperty(), there is a corresponding QObject::property() function. QMetaObject::propertyNames() returns the names of all available properties. QMetaObject::property() returns the property data for a named property: a QMetaProperty object.

Here's a simple example that shows the most important property functions in use:

```
class MyClass : public QObject
```

```
    {
        Q_OBJECT
    public:
        MyClass( QObject * parent=0, const char * name=0 );
        ~MyClass();

        enum Priority { High, Low, VeryHigh, VeryLow };
        void setPriority( Priority );
        Priority priority() const;
    };
```

The class has a property "priority" that is not yet known to the meta object system. In order to make the property known, you have to declare it with the `Q_PROPERTY` macro. The syntax is as follows:

```
Q_PROPERTY( type name READ getFunction [WRITE setFunction]
            [RESET resetFunction] [DESIGNABLE bool]
            [SCRIPTABLE bool] [STORED bool] )
```

For the declaration to be valid, the get function has to be const and to return either the type itself, a pointer to it, or a reference to it. The optional write function has to return void and to take exactly one argument, either the type itself, a pointer or a const reference to it. The meta object compiler enforces this.

The type of a property can be everything QVariant provides or an enumeration type declared in the class itself. Since `MyClass` uses the enumeration type `Priority` for the property, this type has to be registered with the property system as well. This way it will be possible to set a value by name, like this:

```
    obj->setProperty( "priority", "VeryHigh" );
```

Enumeration types are registered with the `Q_ENUMS` macro. Here's the final class declaration including the property related declarations:

```
    class MyClass : public QObject
    {
        Q_OBJECT
        Q_PROPERTY( Priority priority READ priority WRITE setPriority )
        Q_ENUMS( Priority )
    public:
        MyClass( QObject * parent=0, const char * name=0 );
        ~MyClass();

        enum Priority { High, Low, VeryHigh, VeryLow };
        void setPriority( Priority );
        Priority priority() const;
    };
```

Another similar macro is `Q_SETS`. Like `Q_ENUMS`, it registers an enumeration type but marks it in addition as a "set", i.e. the enumeration values can be OR'ed together. An I/O class might have enumeration values "Read" and "Write" and accept "Read|Write": such an enum is best handled with `Q_SETS`, rather than `Q_ENUMS`.

The remaining keywords in the `Q_PROPERTY` section are `RESET`, `DESIGNABLE`, `SCRIPTABLE` and STORED.

`RESET` names a function that will set the property to its default state (which may have changed since initialization). The function must return void and take no arguments.

`DESIGNABLE` declares whether this property is suited for modification by a GUI design tool. The default is `TRUE` for writable properties; otherwise `FALSE`. Instead of `TRUE` or `FALSE`, you can specify a boolean member function.

`SCRIPTABLE` declares whether this property is suited for access by a scripting engine. The default is `TRUE`. Instead of `TRUE` or `FALSE`, you can specify a boolean member function.

`STORED` declares whether the property's value must be remembered when storing an object's state. Stored makes only sense for writable properties. The default value is `TRUE`. Technically superfluous properties (like QPoint pos if QRect geometry is already a property) define this to be `FALSE`.

Connected to the property system is an additional macro, "Q_CLASSINFO", that can be used to attach additional name/value-pairs to a class' meta object, for example:

```
Q_CLASSINFO( "Status", "Very nice class" )
```

Like other meta data, class information is accessible at runtime through the meta object, see QMetaObject::classInfo() for details.

Using the Meta Object Compiler

The Meta Object Compiler, moc among friends, is the program which handles Qt's C++ extensions.

The moc reads a C++ source file. If it finds one or more class declarations that contain the Q_OBJECT macro, it produces another C++ source file which contains the meta object code for the classes that use the Q_OBJECT macro. Among other things, meta object code is required for the signal/slot mechanism, runtime type information and the dynamic property system.

The C++ source file generated by the moc must be compiled and linked with the implementation of the class (or it can be #included into the class's source file).

Using the moc is introduced in chapter 7 of the Qt Tutorial. Chapter 7 includes a simple Makefile that uses the moc and of course source code that uses signals and slots. For more background information on moc, see Why doesn't Qt use templates for signals and slots?.

Usage

The moc is typically used with an input file containing class declarations like this:

```
class MyClass : public QObject
{
    Q_OBJECT
public:
    MyClass( QObject * parent=0, const char * name=0 );
    ~MyClass();

signals:
    void mySignal();

public slots:
    void mySlot();

};
```

In addition to the signals and slots shown above, the moc also implements object properties as in the next example. The Q_PROPERTY macro declares an object property, while Q_ENUMS declares a list of enumeration types within the class to be usable inside the property system. In this particular case we declare a property of the enumeration type `Priority` that is also called "priority" and has a get function `priority()` and a set function `setPriority()`.

```
class MyClass : public QObject
{
    Q_OBJECT
    Q_PROPERTY( Priority priority READ priority WRITE setPriority )
    Q_ENUMS( Priority )
public:
```

```
    MyClass( QObject * parent=0, const char * name=0 );
    ~MyClass();

    enum Priority { High, Low, VeryHigh, VeryLow };
    void setPriority( Priority );
    Priority priority() const;
};
```

Properties can be modified in subclasses with the Q_OVERRIDE macro. The Q_SETS macro declares enums that are to be used as sets, i.e. OR'ed together. Another macro, Q_CLASSINFO, can be used to attach additional name/value-pairs to the class' meta object:

```
class MyClass : public QObject
{
    Q_OBJECT
    Q_CLASSINFO( "Author", "Oscar Peterson")
    Q_CLASSINFO( "Status", "Very nice class")
public:
    MyClass( QObject * parent=0, const char * name=0 );
    ~MyClass();
};
```

The three concepts, signals and slots, properties and class meta-data, can be combined.

The output produced by the moc must be compiled and linked, just as the other C++ code of your program; otherwise the building of your program will fail in the final link phase. By convention, this is done in one of the following two ways:

Method A: The class declaration is found in a header (*.h*) file

If the class declaration above is found in the file *myclass.h*, the moc output should be put in a file called *moc_myclass.cpp*. This file should then be compiled as usual, resulting in an object file *moc_myclass.o* (on Unix) or *moc_myclass.obj* (on Windows). This object should then be included in the list of object files that are linked together in the final building phase of the program.

Method B: The class declaration is found in an implementation (*.cpp*) file

If the class declaration above is found in the file *myclass.cpp*, the moc output should be put in a file called *myclass.moc*. This file should be #included in the implementation file, i.e. *myclass.cpp* should contain the line

```
#include "myclass.moc"
```

after the other code. This will cause the moc-generated code to be compiled and linked together with the normal class definition in *myclass.cpp*, so it is not necessary to compile and link it separately, as in Method A.

Method A is the normal method. Method B can be used in cases where you want the implementation file to be self-contained, or in cases where the Q_OBJECT class is implementation-internal and thus should not be visible in the header file.

Automating moc Usage with Makefiles

For anything but the simplest test programs, it is recommended to automate the running of the moc. By adding some rules to the Makefile of your program, *make* can take care of running moc when necessary and handling the moc output.

We recommend using Trolltech's free makefile generation tool, qmake, for building your Makefiles. This tool recognizes both Method A and B style source files, and generates a Makefile that does all necessary moc handling.

If, on the other hand, you want to build your Makefiles yourself, here are some tips on how to include moc handling.

For Q_OBJECT class declarations in header files, here is a useful makefile rule if you only use GNU make:

```
moc_%.cpp: %.h
        moc $< -o $@
```

If you want to write portably, you can use individual rules of the following form:

```
moc_NAME.cpp: NAME.h
        moc $< -o $@
```

You must also remember to add *moc_NAME.cpp* to your SOURCES (substitute your favorite name) variable and *moc_NAME.o* or *moc_NAME.obj* to your OBJECTS variable.

(While we prefer to name our C++ source files .cpp, the moc doesn't care, so you can use .C, .cc, .CC, .cxx or even .c++ if you prefer.)

For Q_OBJECT class declarations in implementation (.cpp) files, we suggest a makefile rule like this:

```
NAME.o: NAME.moc

NAME.moc: NAME.cpp
        moc -i $< -o $@
```

This guarantees that make will run the moc before it compiles *NAME.cpp*. You can then put

```
#include "NAME.moc"
```

at the end of *NAME.cpp*, where all the classes declared in that file are fully known.

Invoking moc

Here are the command-line options supported by the moc:

-o *file* Write output to *file* rather than to stdout. -f Force the generation of an #include statement in the output. This is the default for files whose name matches the regular expression \.[hH][^.]* (ie. the extension starts with H or h). This option is only useful if you have header files that do not follow the standard naming conventions. -i Do not generate an #include statement in the output. This may be used to run the moc on on a C++ file containing one or more class declarations. You should then #include the meta object code in the .cpp file. If both -i and -f are present, the last one wins. -nw Do not generate any warnings. Not recommended. -ldbg Write a flood of lex debug information to stdout. -p *path* Makes the moc prepend *path/* to the file name in the generated #include statement (if one is generated). -q *path* Makes the moc prepend *path/* to the file name of qt #include files in the generated code.

You can explicitly tell the moc not to parse parts of a header file. It recognizes any C++ comment (//) that contains the substrings MOC_SKIP_BEGIN or MOC_SKIP_END. They work as you would expect and you can have several levels of them. The net result as seen by the moc is as if you had removed all lines between a MOC_SKIP_BEGIN and a MOC_SKIP_END.

Diagnostics

The moc will warn you about a number of dangerous or illegal constructs in the Q_OBJECT class declarations.

If you get linkage errors in the final building phase of your program, saying that YourClass::className() is undefined or that YourClass lacks a vtbl, something has been done wrong. Most often, you have forgotten to compile or #include the moc-generated C++ code, or (in the former case) include that object file in the link command.

Limitations

The moc does not expand #include or #define, it simply skips any preprocessor directives it encounters. This is regrettable, but is normally not a problem in practice.

The moc does not handle all of C++. The main problem is that class templates cannot have signals or slots. Here is an example:

```
class SomeTemplate : public QFrame {
    Q_OBJECT
    ...
signals:
    void bugInMocDetected( int );
};
```

Less importantly, the following constructs are illegal. All of them have alternatives which we think are usually better, so removing these limitations is not a high priority for us.

Multiple inheritance requires QObject to be first

If you are using multiple inheritance, moc assumes that the *first* inherited class is a subclass of QObject. Also, be sure that *only* the first inherited class is a QObject.

```
class SomeClass : public QObject, public OtherClass {
    ...
};
```

(This limitation is almost impossible to remove; since the moc does not expand #include or #define, it cannot find out which one of the base classes is a QObject.)

Function pointers cannot be arguments to signals or slots

In most cases where you would consider using function pointers as signal/slot arguments, we think inheritance is a better alternative. Here is an example of illegal syntax:

```
class SomeClass : public QObject {
    Q_OBJECT
    ...
public slots:
    // illegal
    void apply( void (*apply)(List *, void *), char * );
};
```

You can work around this restriction like this:

```
typedef void (*ApplyFunctionType)( List *, void * );

class SomeClass : public QObject {
    Q_OBJECT
    ...
public slots:
    void apply( ApplyFunctionType, char * );
};
```

It may sometimes be even better to replace the function pointer with inheritance and virtual functions, signals or slots.

Friend declarations cannot be placed in signals or slots sections

Sometimes it will work, but in general, friend declarations cannot be placed in signals or slots sections. Put them in the private, protected or public sections instead. Here is an example of the illegal syntax:

```
class SomeClass : public QObject {
    Q_OBJECT
    ...
signals:
    friend class ClassTemplate; // WRONG
};
```

Signals and slots cannot be upgraded

The C++ feature of upgrading an inherited member function to public status is not extended to cover signals and slots. Here is an illegal example:

```
class Whatever : public QButtonGroup {
    ...
public slots:
    void QButtonGroup::buttonPressed; // WRONG
    ...
};
```

The QButtonGroup::buttonPressed() slot is protected.

C++ quiz: What happens if you try to upgrade a protected member function which is overloaded?

1. All the functions are overloaded.
2. That is not legal C++.

Type macros cannot be used for signal and slot parameters

Since the moc does not expand #define, type macros that take an argument will not work in signals and slots. Here is an illegal example:

```
#ifdef ultrix
#define SIGNEDNESS(a) unsigned a
#else
#define SIGNEDNESS(a) a
#endif

class Whatever : public QObject {
    ...
signals:
    void someSignal( SIGNEDNESS(int) );
    ...
};
```

A #define without parameters will work as expected.

Nested classes cannot be in the signals or slots sections nor have signals or slots

Here's an example:

```
class A {
    Q_OBJECT
public:
    class B {
    public slots:   // WRONG
        void b();
        ...
    };
signals:
    class B {       // WRONG
        void b();
        ...
    }:
};
```

Constructors cannot be used in signals or slots sections

It is a mystery to us why anyone would put a constructor in either the signals or slots sections. You can't anyway (except that it happens to work in some cases). Put them in private, protected or public sections, where they belong. Here is an example of the illegal syntax:

```
class SomeClass : public QObject {
    Q_OBJECT
public slots:
    SomeClass( QObject *parent, const char *name )
        : QObject( parent, name ) { } // WRONG
    ...
};
```

Properties need to be declared before the public section that contains the respective get and set functions

Declaring the first property within or after the public section that contains the type definition and the respective get and set functions does not work as expected. The moc will complain that it can neither find the functions nor resolve the type. Here is an example of the illegal syntax:

```
class SomeClass : public QObject {
    Q_OBJECT
public:
    ...
    Q_PROPERTY( Priority priority READ priority WRITE setPriority ) // WRONG
    Q_ENUMS( Priority ) // WRONG
    enum Priority { High, Low, VeryHigh, VeryLow };
    void setPriority( Priority );
    Priority priority() const;
    ...
};
```

Work around this limitation by declaring all properties at the beginning of the class declaration, right after Q_OBJECT:

```
class SomeClass : public QObject {
    Q_OBJECT
    Q_PROPERTY( Priority priority READ priority WRITE setPriority )
    Q_ENUMS( Priority )
public:
    ...
    enum Priority { High, Low, VeryHigh, VeryLow };
    void setPriority( Priority );
    Priority priority() const;
    ...
};
```

Signals and Slots

Signals and slots are used for communication between objects. The signal/slot mechanism is a central feature of Qt and probably the part that differs most from other toolkits.

In GUI programming we often want a change in one widget to be notified to another widget. More generally, we want objects of any kind to be able to communicate with one another. For example if we were parsing an XML file we might want to notify a list view that we're using to represent the XML file's structure whenever we encounter a new tag.

Older toolkits achieve this kind of communication using callbacks. A callback is a pointer to a function, so if you want a processing function to notify you about some event you pass a pointer to another function (the callback) to the processing function. The processing function then calls the callback when appropriate. Callbacks have two fundamental flaws. Firstly they are not type safe. We can never be certain that the processing function will call the callback with the correct arguments. Secondly the callback is strongly coupled to the processing function since the processing function must know which callback to call.

In Qt we have an alternative to the callback technique. We use signals and slots. A signal is emitted when a particular event occurs. Qt's widgets have many pre-defined signals, but we can always subclass to add our own. A slot is a function that is called in reponse to a particular signal. Qt's widgets have many pre-defined slots, but it is common practice to add your own slots so that you can handle the signals that you are interested in.

The signals and slots mechanism is type safe: the signature of a signal must match the signature of the receiving slot. (In fact a slot may have a shorter signature than the signal it receives because it can ignore extra arguments.) Since the signatures are compatible, the compiler can help us detect type mismatches. Signals and slots are loosely coupled: a class which emits a signal neither knows nor cares which slots receive the signal. Qt's signals and slots mechanism ensures that if you connect a signal to a slot, the slot will be called with the signal's parameters at the right time. Signals and slots can take any number of arguments of any type. They are completely typesafe: no more callback core dumps!

All classes that inherit from QObject or one of its subclasses (e.g. QWidget) can contain signals and slots. Signals are emitted by objects when they change their state in a way that may be interesting to the outside world. This is all the object does to communicate. It does not know or care whether anything is receiving the signals it emits. This is true information encapsulation, and ensures that the object can be used as a software component.

Slots can be used for receiving signals, but they are normal member functions. A slot does not know if it has any signals connected to it. Again, the object does not know about the communication mechanism and can be used as a true software component.

You can connect as many signals as you want to a single slot, and a signal can be connected to as many slots as you desire. It is even possible to connect a signal directly to another signal. (This will emit the second signal immediately whenever the first is emitted.)

Together, signals and slots make up a powerful component programming mechanism.

A Small Example

A minimal C++ class declaration might read:

```
class Foo
{
public:
    Foo();
    int value() const { return val; }
    void setValue( int );
private:
    int val;
};
```

A small Qt class might read:

```
class Foo : public QObject
{
    Q_OBJECT
public:
    Foo();
    int value() const { return val; }
public slots:
    void setValue( int );
signals:
    void valueChanged( int );
private:
    int val;
};
```

This class has the same internal state, and public methods to access the state, but in addition it has support for component programming using signals and slots: this class can tell the outside world that its state has changed by emitting a signal, `valueChanged()`, and it has a slot which other objects may send signals to.

All classes that contain signals and/or slots must mention Q_OBJECT in their declaration.

Slots are implemented by the application programmer. Here is a possible implementation of Foo::setValue():

```
void Foo::setValue( int v )
{
    if ( v != val ) {
        val = v;
        emit valueChanged(v);
    }
}
```

The line `emit valueChanged(v)` emits the signal `valueChanged` from the object. As you can see, you emit a signal by using `emit signal(arguments)`.

Here is one way to connect two of these objects together:

```
Foo a, b;
connect(&a, SIGNAL(valueChanged(int)), &b, SLOT(setValue(int)));
b.setValue( 11 ); // a == undefined  b == 11
a.setValue( 79 ); // a == 79         b == 79
b.value();
```

Calling `a.setValue(79)` will make `a` emit a `valueChanged()` signal, which `b` will receive in its `setValue()` slot, i.e. `b.setValue(79)` is called. `b` will then, in turn, emit the same `valueChanged()` signal, but since no slot has been connected to `b`'s `valueChanged()` signal, nothing happens (the signal disappears).

Note that the `setValue()` function sets the value and emits the signal only if `v != val`. This prevents infinite looping in the case of cyclic connections (e.g. if `b.valueChanged()` were connected to `a.setValue()`).

This example illustrates that objects can work together without knowing each other, as long as there is someone around to set up a connection between them initially.

The preprocessor changes or removes the `signals`, `slots` and `emit` keywords so that the compiler is presented with standard C++.

Run the moc on class definitions that contain signals or slots. This produces a C++ source file which should be compiled and linked with the other object files for the application.

Signals

Signals are emitted by an object when its internal state has changed in some way that might be interesting to the object's client or owner. Only the class that defines a signal and its subclasses can emit the signal.

A list box, for example, emits both `highlighted()` and `activated()` signals. Most objects will probably only be interested in `activated()` but some may want to know about which item in the list box is currently highlighted. If the signal is interesting to two different objects you just connect the signal to slots in both objects.

When a signal is emitted, the slots connected to it are executed immediately, just like a normal function call. The signal/slot mechanism is totally independent of any GUI event loop. The `emit` will return when all slots have returned.

If several slots are connected to one signal, the slots will be executed one after the other, in an arbitrary order, when the signal is emitted.

Signals are automatically generated by the moc and must not be implemented in the .cpp file. They can never have return types (i.e. use `void`).

A note about arguments. Our experience shows that signals and slots are more reusable if they do *not* use special types. If QScrollBar::valueChanged() were to use a special type such as the hypothetical QRangeControl::Range, it could only be connected to slots designed specifically for QRangeControl. Something as simple as the program in Tutorial 5 would be impossible.

Slots

A slot is called when a signal connected to it is emitted. Slots are normal C++ functions and can be called normally; their only special feature is that signals can be connected to them. A slot's arguments cannot have default values, and, like signals, it is rarely wise to use your own custom types for slot arguments.

Since slots are normal member functions with just a little extra spice, they have access rights like ordinary member functions. A slot's access right determines who can connect to it:

A `public slots:` section contains slots that anyone can connect signals to. This is very useful for component programming: you create objects that know nothing about each other, connect their signals and slots so that information is passed correctly, and, like a model railway, turn it on and leave it running.

A `protected slots:` section contains slots that this class and its subclasses may connect signals to. This is intended for slots that are part of the class' implementation rather than its interface to the rest of the world.

A `private slots:` section contains slots that only the class itself may connect signals to. This is intended for very tightly connected classes, where even subclasses aren't trusted to get the connections right.

You can also define slots to be virtual, which we have found quite useful in practice.

The signals and slots mechanism is efficient, but not quite as fast as "real" callbacks. Signals and slots are slightly slower because of the increased flexibility they provide, although the difference for real applications is insignificant. In general, emitting a signal that is connected to some slots, is approximately ten times slower than calling the

receivers directly, with non-virtual function calls. This is the overhead required to locate the connection object, to safely iterate over all connections (i.e. checking that subsequent receivers have not been destroyed during the emission) and to marshall any parameters in a generic fashion. While ten non-virtual function calls may sound like a lot, it's much less overhead than any 'new' or 'delete' operation, for example. As soon as you perform a string, vector or list operation that behind the scene requires 'new' or 'delete', the signals and slots overhead is only responsible for a very small proportion of the complete function call costs. The same is true whenever you do a system call in a slot - or indirectly call more than ten functions. On an i586-500, you can emit around 2,000,000 signals per second connected to one receiver, or around 1,200,000 per second connected to two receivers. The simplicity and flexibility of the signals and slots mechanism is well worth the overhead, which your users won't even notice.

Meta Object Information

The meta object compiler (moc) parses the class declaration in a C++ file and generates C++ code that initializes the meta object. The meta object contains names of all signal and slot members, as well as pointers to these functions. (For more information on Qt's Meta Object System, see Why doesn't Qt use templates for signals and slots?.)

The meta object contains additional information such as the object's class name. You can also check if an object inherits a specific class, for example:

```
if ( widget->inherits("QButton") ) {
      // yes, it is a push button, radio button etc.
}
```

A Real Example

Here is a simple commented example (code fragments from qlcdnumber.h).

```
#include "qframe.h"
#include "qbitarray.h"

class QLCDNumber : public QFrame
```

QLCDNumber inherits QObject, which has most of the signal/slot knowledge, via QFrame and QWidget, and #include's the relevant declarations.

```
{
    Q_OBJECT
```

Q_OBJECT is expanded by the preprocessor to declare several member functions that are implemented by the moc; if you get compiler errors along the lines of "virtual function QButton::className not defined" you have probably forgotten to run the moc or to include the moc output in the link command.

```
public:
    QLCDNumber( QWidget *parent=0, const char *name=0 );
    QLCDNumber( uint numDigits, QWidget *parent=0, const char *name=0 );
```

It's not obviously relevant to the moc, but if you inherit QWidget you almost certainly want to have the *parent* and *name* arguments in your constructors, and pass them to the parent constructor.

Some destructors and member functions are omitted here; the moc ignores member functions.

```
signals:
    void    overflow();
```

QLCDNumber emits a signal when it is asked to show an impossible value.

If you don't care about overflow, or you know that overflow cannot occur, you can ignore the overflow() signal, i.e. don't connect it to any slot.

If, on the other hand, you want to call two different error functions when the number overflows, simply connect the signal to two different slots. Qt will call both (in arbitrary order).

```
public slots:
    void    display( int num );
    void    display( double num );
    void    display( const char *str );
    void    setHexMode();
    void    setDecMode();
    void    setOctMode();
    void    setBinMode();
    void    smallDecimalPoint( bool );
```

A slot is a receiving function, used to get information about state changes in other widgets. QLCDNumber uses it, as the code above indicates, to set the displayed number. Since `display()` is part of the class' interface with the rest of the program, the slot is public.

Several of the example programs connect the newValue signal of a QScrollBar to the display slot, so the LCD number continuously shows the value of the scroll bar.

Note that display() is overloaded; Qt will select the appropriate version when you connect a signal to the slot. With callbacks, you'd have to find five different names and keep track of the types yourself.

Some irrelevant member functions have been omitted from this example.

```
};
```

Why doesn't Qt use templates for signals and slots?

A simple answer is that when Qt was designed, it was not possible to fully exploit the template mechanism in multi-platform applications due to the inadequacies of various compilers. Even today, many widely used C++ compilers have problems with advanced templates. For example, you cannot safely rely on partial template instantiation, which is essential for some non-trivial problem domains. Thus Qt's usage of templates has to be rather conservative. Keep in mind that Qt is a multi-platform toolkit, and progress on the Linux/g++ platform does not necessarily improve the situation elsewhere.

Eventually those compilers with weak template implementations will improve. But even if all our users had access to a fully standards compliant modern C++ compiler with excellent template support, we would not abandon the string-based approach used by our meta object compiler. Here are five reasons why:

1. Syntax matters

Syntax isn't just sugar: the syntax we use to express our algorithms can significantly affect the readability and maintainability of our code. The syntax used for Qt's signals and slots has proved very successful in practice. The syntax is intuitive, simple to use and easy to read. People learning Qt find the syntax helps them understand and utilize the signals and slots concept — despite its highly abstract and generic nature. Furthermore, declaring signals in class definitions ensures that the signals are protected in the sense of protected C++ member functions. This helps programmers get their design right from the very beginning, without even having to think about design patterns.

2. Precompilers are good

Qt's `moc` (Meta Object Compiler) provides a clean way to go beyond the compiled language's facilities. It does so by generating additional C++ code which can be compiled by any standard C++ compiler. The `moc` reads C++ source files. If it finds one or more class declarations that contain the "Q_OBJECT" macro, it produces another C++ source file which contains the meta object code for those classes. The C++ source file generated by the `moc` must be compiled and linked with the implementation of the class (or it can be `#included` into the class's source file). Typically `moc` is not called manually, but automatically by the build system, so it requires no additional effort by the programmer.

There are other precompilers, for example, `rpc` and `idl`, that enable programs or objects to communicate over process or machine boundaries. The alternatives to precompilers are hacked compilers, proprietary languages or graphical programming tools with dialogs or wizards that generate obscure code. Rather than locking our customers into a proprietary C++ compiler or into a particular Integrated Development Environment, we enable them to use whatever tools they prefer. Instead of forcing programmers to add generated code into source repositories, we encourage them to add our tools to their build system: cleaner, safer and more in the spirit of UNIX.

3. Flexibility is king

C++ is a standarized, powerful and elaborate general-purpose language. It's the only language that is exploited on such a wide range of software projects, spanning every kind of application from entire operating systems, database servers and high end graphics applications to common desktop applications. One of the keys to C++'s success is its scalable language design that focuses on maximum performance and minimal memory consumption whilst still maintaining ANSI-C compatibility.

For all these advantages, there are some downsides. For C++, the static object model is a clear disadvantage over the dynamic messaging approach of Objective C when it comes to component-based graphical user interface programming. What's good for a high end database server or an operating system isn't necessarily the right design choice for a GUI frontend. With `moc`, we have turned this disadvantage into an advantage, and added the flexibility required to meet the challenge of safe and efficient graphical user interface programming.

Our approach goes far beyond anything you can do with templates. For example, we can have object properties. And we can have overloaded signals and slots, which feels natural when programming in a language where overloads are a key concept. Our signals add zero bytes to the size of a class instance, which means we can add new signals without breaking binary compatibility. Because we do not rely on excessive inlining as done with templates, we can keep the code size smaller. Adding new connections just expands to a simple function call rather than a complex template function.

Another benefit is that we can explore an object's signals and slots at runtime. We can establish connections using type-safe call-by-name, without having to know the exact types of the objects we are connecting. This is impossible with a template based solution. This kind of runtime introspection opens up new possibilities, for example GUIs that are generated and connected from Qt Designer's XML `ui` files.

4. Calling performance is not everything

Qt's signals and slots implementation is not as fast as a template-based solution. While emitting a signal is approximately the cost of four ordinary function calls with common template implementations, Qt requires effort comparable to about ten function calls. This is not surprising since the Qt mechanism includes a generic marshaller, introspection and ultimately scriptability. It does not rely on excessive inlining and code expansion and it provides unmatched runtime safety. Qt's iterators are safe while those of faster template-based systems are not. Even during the process of emitting a signal to several receivers, those receivers can be deleted safely without your program crashing. Without this safety, your application would eventually crash with a difficult to debug free'd memory read or write error.

Nonetheless, couldn't a template-based solution improve the performance of an application using signals and slots? While it is true that Qt adds a small overhead to the cost of calling a slot through a signal, the cost of the call is only a small proportion of the entire cost of a slot. Benchmarking against Qt's signals and slots system is typically done with empty slots. As soon as you do anything useful in your slots, for example a few simple string operations, the calling overhead becomes negligible. Qt's system is so optimized that anything that requires operator new or delete (for example, string operations or inserting/removing something from a template container) is significantly more expensive than emitting a signal.

Aside: If you have a signals and slots connection in a tight inner loop of a performance critical task and you identify this connection as the bottleneck, think about using the standard listener-interface pattern rather than signals and slots. In cases where this occurs, you probably only require a 1:1 connection anyway. For example, if you have an object that downloads data from the network, it's a perfectly sensible design to use a signal to indicate that the requested data arrived. But if you need to send out every single byte one by one to a consumer, use a listener interface rather than signals and slots.

5. No limits

Because we had the `moc` for signals and slots, we could add other useful things to it that could not not be done with templates. Among these are scoped translations via a generated `tr()` function, and an advanced property system with introspection and extended runtime type information. The property system alone is a great advantage:

a powerful and generic user interface design tool like Qt Designer would be a lot harder to write - if not impossible - without a powerful and introspective property system.

C++ with the `moc` preprocessor essentially gives us the flexibility of Objective-C or of a Java Runtime Environment, while maintaining C++'s unique performance and scalability advantages. It is what makes Qt the flexible and comfortable tool we have today.

Timers

QObject, the base class of all Qt objects, provides the basic timer support in Qt. With QObject::startTimer(), you start a timer with an *interval* in milliseconds as argument. The function returns a unique integer timer id. The timer will now "fire" every *interval* milliseconds, until you explicitly call QObject::killTimer() with the timer id.

For this mechanism to work, the application has to run in an event loop. You start an event loop with QApplication::exec(). When a timer fires, the application sends a QTimerEvent, and the flow of control leaves the event loop until the timer event is processed. This implies that a timer cannot fire while your application is busy doing something else. In other words: the accuracy of timers depends on the granularity of your application.

There is practically no upper limit for the interval value (more than one year is possible). The accuracy depends on the underlying operating system. Windows 95/98 has 55 millisecond (18.2 times per second) accuracy; other systems that we have tested (UNIX X11, Windows NT and OS/2) can handle 1 millisecond intervals.

The main API for the timer functionality is QTimer. That class provides regular timers that emit a signal when the timer fires, and inherit QObject so that it fits well into the ownership structure of most GUI programs. The normal way of using it is like this:

```
QTimer * counter = new QTimer( this );
connect( counter, SIGNAL(timeout()),
         this, SLOT(updateCaption()) );
counter->start( 1000 );
```

The counter timer is made into a child of this widget, so that when this widget is deleted, the timer is deleted too. Next, its timeout signal is connected to the slot that will do the work, and finally it's started.

QTimer also provides a simple one-shot timer API. QButton uses it to show the button being pressed down and then (0.1 second later) be released when the keyboard is used to "press" a button, for example:

```
QTimer::singleShot( 100, this, SLOT(animateTimeout()) );
```

0.1 seconds after this line of code is executed, the same button's animateTimeout() slot is called.

Here is an outline of a slightly longer example that combines object communication via signals and slots with a QTimer object. It demonstrates how to use timers to perform intensive calculations in a single-threaded application without blocking the user interface.

```
// The Mandelbrot class uses a QTimer to calculate the mandelbrot
// set one scanline at a time without blocking the CPU.  It
// inherits QObject to use signals and slots.  Calling start()
// starts the calculation. The done() signal is emitted when it
// has finished.  Note that this example is not complete, just an
// outline.

class Mandelbrot : public QObject
{
    Q_OBJECT // required for signals/slots
public:
```

```
    Mandelbrot( QObject *parent=0, const char *name );
    ...
public slots:
    void start();
signals:
    void done();
private slots:
    void calculate();
    private:
    QTimer timer;
    ...
};

//
// Constructs and initializes a Mandelbrot object.
//

Mandelbrot::Mandelbrot( QObject *parent=0, const char *name )
: QObject( parent, name )
{
    connect( &timer, SIGNAL(timeout()), SLOT(calculate()) );
    ...
}

//
// Starts the calculation task. The internal calculate() slot
// will be activated every 10 milliseconds.
//

void Mandelbrot::start()
{
    if ( !timer.isActive() ) // not already running
        timer.start( 10 );   // timeout every 10 ms
}

//
// Calculates one scanline at a time.
// Emits the done() signal when finished.
//

void Mandelbrot::calculate()
{
    ...                // perform the calculation for a scanline
    if ( finished ) {  // no more scanlines
        timer.stop();
        emit done();
    }
}
```

Debugging Techniques

Here we present some useful hints to debugging your Qt-based software.

Command Line Options

When you run a Qt program you can specify several command line options that can help with debugging.

- -nograb The application should never grab the mouse or the keyboard. This option is set by default when the program is running in the gdb debugger under Linux.
- -dograb Ignore any implicit or explicit -nograb. -dograb wins over -nograb even when -nograb is last on the command line.
- -sync Runs the application in X synchronous mode. Synchronous mode forces the X server to perform each X client request immediately and not use buffer optimization. It makes the program easier to debug and often much slower. The -sync option is only valid for the X11 version of Qt.

Warning and Debugging Messages

Qt includes three global functions for writing out warning and debug text.

- qDebug() for writing debug output for testing etc.
- qWarning() for writing warning output when program errors occur.
- qFatal() for writing fatal error messages and exit.

The Qt implementation of these functions prints the text to the stderr output under Unix/X11 and to the debugger under Windows. You can take over these functions by installing a message handler; qInstallMsgHandler().

The debugging functions QObject::dumpObjectTree() and QObject::dumpObjectInfo() are often useful when an application looks or acts strangely. More useful if you use object names than not, but often useful even without names.

Debugging Macros

The header file qglobal.h contains many debugging macros and #defines.

Two important macros are:

- Q_ASSERT(b) where b is a boolean expression, writes the warning: "ASSERT: 'b' in file file.cpp (234)" if b is FALSE.
- Q_CHECK_PTR(p) where p is a pointer. Writes the warning "In file file.cpp, line 234: Out of memory" if p is null.

These macros are useful for detecting program errors, e.g. like this:

```
char *alloc( int size )
{
    Q_ASSERT( size > 0 );
    char *p = new char[size];
    Q_CHECK_PTR( p );
    return p;
}
```

If you define the flag QT_FATAL_ASSERT, Q_ASSERT will call fatal() instead of warning(), so a failed assertion will cause the program to exit after printing the error message.

Note that the Q_ASSERT macro is a null expression if `QT_CHECK_STATE` (see below) is not defined. Any code in it will simply not be executed. Similarly Q_CHECK_PTR is a null expression if `QT_CHECK_NULL` is not defined. Here is an example of how you should *not* use Q_ASSERT and Q_CHECK_PTR:

```
char *alloc( int size )
{
    char *p;
    Q_CHECK_PTR( p = new char[size] ); // WRONG
    return p;
}
```

The problem is tricky: *p* is set to a sane value only as long as the correct checking flags are defined. If this code is compiled without the QT_CHECK_NULL flag defined, the code in the Q_CHECK_PTR expression is not executed (correctly, since it's only a debugging aid) and *alloc* returns a wild pointer.

The Qt library contains hundreds of internal checks that will print warning messages when some error is detected.

The tests for sanity and the resulting warning messages inside Qt are conditional, based on the state of various debugging flags:

- QT_CHECK_STATE: Check for consistent/expected object state
- QT_CHECK_RANGE: Check for variable range errors
- QT_CHECK_NULL: Check for dangerous null pointers
- QT_CHECK_MATH: Check for dangerous math, e.g. division by 0
- QT_NO_CHECK: Turn off all QT_CHECK_... flags
- QT_DEBUG: Enable debugging code
- QT_NO_DEBUG: Turn off QT_DEBUG flag

By default, both QT_DEBUG and all the QT_CHECK flags are on. To turn off QT_DEBUG, define QT_NO_DEBUG. To turn off the QT_CHECK flags, define QT_NO_CHECK.

Example:

```
void f( char *p, int i )
{
#if defined(QT_CHECK_NULL)
    if ( p == 0 )
        qWarning( "f: Null pointer not allowed" );
#endif

#if defined(QT_CHECK_RANGE)
    if ( i < 0 )
        qWarning( "f: The index cannot be negative" );
#endif
}
```

Common bugs

There is one bug that is so common that it deserves mention here: If you include the Q_OBJECT macro in a class declaration and run the *moc*, but forget to link the moc-generated object code into your executable, you will get very confusing error message.

Any link error complaining about a lack of `vtbl`, `_vtbl`, `__vtbl` or similar is likely to be this problem.

Class Inheritance Hierarchy

This list shows the C++ class inheritance relations between the classes in the Qt API.

The list is sorted roughly, but not completely, alphabetically.

- QAccessible
  - QAccessibleInterface
- QAsciiCache
- QAsciiCacheIterator
- QAsciiDictIterator
- QAsyncIO
  - QDataSink
  - QDataSource
    - QIODeviceSource
- QBitVal
- QCacheIterator
- QCanvasPixmapArray
- QChar
- QCharRef
- QColor
- QColorGroup
- QConstString
- QDataStream
- QDate
- QDateEdit
- QDateTime
- QDictIterator
- QDir
- QDomImplementation
- QDomNamedNodeMap
- QDomNode
  - QDomAttr
  - QDomCharacterData
    - QDomComment
    - QDomText
      - QDomCDATASection

  - QDomDocument
  - QDomDocumentFragment
  - QDomDocumentType
  - QDomElement
  - QDomEntity
  - QDomEntityReference
  - QDomNotation
  - QDomProcessingInstruction

- QDomNodeList
- QDropSite
- QFileInfo
- QFilePreview
- QFocusData
- QFont
- QFontDatabase
- QFontInfo
- QFontManager
- QFontMetrics
- QGL
  - QGLContext
  - QGLFormat
- QGLayoutIterator
- QGLColormap
- QGuardedPtr
- QHostAddress
- QIconDragItem
- QIconSet
- QImage
- QImageConsumer
- QImageDecoder
- QImageFormat
- QImageFormatPlugin
- QImageFormatType
- QImageIO
- QIntCache
- QIntCacheIterator
- QIntDictIterator
- QIODevice
  - QBuffer
  - QFile
  - QSocketDevice
- QLayoutItem
  - QSpacerItem
  - QWidgetItem

- QLayoutIterator
- QLibrary
- QListBoxItem
  - QListBoxPixmap
  - QListBoxText
- QListViewItemIterator
- QLock
- QMap
- QMapConstIterator
- QMapIterator
- QMemArray
  - QByteArray
    - QBitArray
    - QCString
  - QPointArray
- QMenuData
- QMetaObject
- QMetaProperty
- QMimeSource
- QMimeSourceFactory
- QMovie
- QMutex
- QNPlugin
- QNPStream
- QPaintDevice
  - QPicture
  - QPixmap
    - QBitmap
    - QCanvasPixmap
  - QPrinter
- QPaintDeviceMetrics
- QPair
- QPalette
- QPixmapCache
- QPNGImagePacker
- QPoint
- QPtrCollection
  - QAsciiDict
  - QCache
  - QDict
  - QIntDict
  - QPtrDict
  - QPtrList

        * QSortedList
        * QStrList
            · QStrIList
    - QPtrVector

- QPtrDictIterator
- QPtrListIterator
    - QStrListIterator
- QPtrQueue
- QPtrStack
- QRangeControl
- QRect
- QRegExp
- QRegion
- QScreen
- QSemaphore
- QSettings
- QSimpleRichText
- QSize
- QSizePolicy
- QSql
- QSqlDriverPlugin
- QSqlError
- QSqlField
- QSqlFieldInfo
- QSqlPropertyMap
- QSqlQuery
- QSqlRecord
    - QSqlCursor
    - QSqlIndex
- QSqlRecordInfo
- QSqlResult
- QString
- QStyleFactory
- QStyleOption
- QStylePlugin
- Qt
    - QBrush
    - QCanvasItem
        * QCanvasPolygonalItem
            · QCanvasEllipse
            · QCanvasLine
            · QCanvasPolygon
            · QCanvasSpline
            · QCanvasRectangle

    * QCanvasSprite
    * QCanvasText
  - QCursor
  - QCustomMenuItem
  - QEvent
    * QChildEvent
    * QCloseEvent
    * QContextMenuEvent
    * QCustomEvent
    * QDragLeaveEvent
    * QDropEvent
      · QDragMoveEvent
      · QDragEnterEvent
    * QFocusEvent
    * QHideEvent
    * QIMEvent
    * QKeyEvent
    * QMouseEvent
    * QMoveEvent
    * QPaintEvent
    * QResizeEvent
    * QShowEvent
    * QTabletEvent
    * QTimerEvent
    * QWheelEvent
  - QIconViewItem
  - QKeySequence
  - QListViewItem
    * QCheckListItem
  - QObject
    * QAccel
    * QAccessibleObject
    * QAction
      · QActionGroup
    * QApplication
      · QXtApplication
    * QCanvas
    * QClipboard
    * QCopChannel
    * QDataPump
    * QDns
    * QDragObject
      · QIconDrag
      · QImageDrag

- · QStoredDrag
- · QColorDrag
- · QUriDrag
- · QTextDrag

* QEditorFactory
    - · QSqlEditorFactory
* QFileIconProvider
* QLayout
    - · QBoxLayout
    - · QHBoxLayout
    - · QVBoxLayout
    - · QGridLayout
* QNetworkOperation
* QNetworkProtocol
    - · QFtp
    - · QHttp
    - · QLocalFs
* QNPInstance
* QObjectCleanupHandler
* QProcess
* QServerSocket
* QSessionManager
* QSignal
* QSignalMapper
* QSocket
* QSocketNotifier
* QSound
* QSqlDatabase
* QSqlDriver
* QSqlForm
* QStyle
    - · QCommonStyle
    - · QMotifStyle
    - · QCDEStyle
    - · QMotifPlusStyle
    - · QSGIStyle
    - · QWindowsStyle
    - · QAquaStyle
    - · QPlatinumStyle

* QStyleSheet
* QTimer
* QToolTipGroup
* QTranslator
* QUrlOperator
* QValidator
    - QDoubleValidator
    - QIntValidator
    - QRegExpValidator
* QWidget
    - QButton
    - QCheckBox
    - QPushButton
    - QRadioButton
    - QToolButton
    - QComboBox
    - QDataBrowser
    - QDataView
    - QDateTimeEdit
    - QDesktopWidget
    - QDial
    - QDialog
    - QColorDialog
    - QErrorMessage
    - QFileDialog
    - QFontDialog
    - QInputDialog
    - QMessageBox
    - QProgressDialog
    - QTabDialog
    - QWizard
    - QDockArea
    - QFrame
    - QDockWindow
    - QToolBar
    - QGrid
    - QGroupBox
    - QButtonGroup
    - QHButtonGroup
    - QVButtonGroup

- QHGroupBox
- QVGroupBox

- QHBox
- QVBox

- QLabel
- QLCDNumber
- QLineEdit
- QMenuBar
- QPopupMenu
- QProgressBar
- QScrollView
- QCanvasView
- QGridView
- QIconView
- QListBox
- QListView
- QTable
- QDataTable

- QTextEdit
- QMultiLineEdit
- QTextBrowser
- QTextView

- QSplitter
- QtTableView
- QtMultiLineEdit

- QWidgetStack

- QGLWidget
- QHeader
- QMainWindow
- QNPWidget
- QScrollBar
- QSizeGrip
- QSlider
- QSpinBox
- QStatusBar
- QTabBar
- QTabWidget
- QWorkspace
- QXtWidget

* QWSKeyboardHandler

    * QWSMouseHandler

  - QPainter
  - QPen
  - QStyleSheetItem
  - QTab
  - QTableItem
    * QCheckTableItem
    * QComboTableItem
  - QThread
  - QToolTip
  - QWhatsThis

- QTableSelection
- QTextCodec
  - QEucJpCodec
  - QEucKrCodec
  - QGbkCodec
  - QHebrewCodec
  - QJisCodec
  - QSjisCodec
  - QTsciiCodec
- QTextCodecPlugin
- QTextDecoder
- QTextEncoder
- QTextStream
  - QTextIStream
  - QTextOStream
- QTime
- QTimeEdit
- QTranslatorMessage
- QUrl
- QUrlInfo
- QValueList
  - QCanvasItemList
  - QStringList
  - QValueStack
- QValueListConstIterator
- QValueListIterator
- QValueVector
- QVariant
- QWaitCondition
- QWidgetFactory

- QWidgetPlugin
- QWindowsMime
- QWMatrix
- QWSDecoration
- QWSServer
- QWSWindow
- QXmlAttributes
- QXmlContentHandler
  - QXmlDefaultHandler
- QXmlDeclHandler
- QXmlDTDHandler
- QXmlEntityResolver
- QXmlErrorHandler
- QXmlInputSource
- QXmlLexicalHandler
- QXmlLocator
- QXmlNamespaceSupport
- QXmlParseException
- QXmlReader
  - QXmlSimpleReader

The Feature Definition File

The file `src/tools/qfeatures.h` includes the file `src/tools/qconfig.h`. By modifying `qconfig.h`, you can define a subset of the full Qt functionality that you wish to have available on your installation.

Note that such modification is only supported on Qt/Embedded platforms, where reducing the size of Qt is important and the application-set is often fixed.

The `config.h` definition file simply defines macros to disable features. Some features are dependent on other features and these dependencies are expressed in `qfeatures.h`.

The available options are:

Macro	Disables	Set automatically by
	Images (QImageIO)	
QT_NO_IMAGEIO_BMP	The Microsoft Bitmap image file format.	
QT_NO_IMAGEIO_PPM	The Portable Pixmap image file format.	
QT_NO_IMAGEIO_XBM	The X11 Bitmap image file format.	
QT_NO_IMAGEIO_XPM	The X11 Pixmap image file format.	
QT_NO_IMAGEIO_PNG	The Portable Network Graphics image file format.	
	Animation	
QT_NO_ASYNC_IO	Asynchronous I/O (QAsyncIO)	
QT_NO_ASYNC_IMAGE_IO	Asynchronous Image I/O and GIF image support (QImageDecoder, ...)	
QT_NO_MOVIE	Animation support (QMovie)	QT_NO_ASYNC_IO, QT_NO_ASYNC_IMAGE_IO
	Fonts	
QT_NO_TRUETYPE	TrueType (TTF and TTC) font file format, only used by Qt/Embedded.	
QT_NO_BDF	Bitmap Distribution Format (BDF) font file format, only used by Qt/Embedded.	
QT_NO_FONTDATABASE	Font database.	
	Internationalization	
QT_NO_I18N	Conversions between Unicode and 8-bit encodings.	
QT_NO_UNICODETABLES	Large tables defining such things as upper and lowercase conversions for all Unicode characters.	
	MIME	
QT_NO_MIME	Multipurpose Internet Mail Extensions, an internet standard for encoding and tagging typed data (eg. text, images, colors) (QMimeSource)	
QT_NO_RICHTEXT	HTML-like text (QStyleSheet, QLabel)	QT_NO_MIME
QT_NO_DRAGANDDROP	Drag-and-drop data between applications (QDragObject)	QT_NO_MIME
QT_NO_CLIPBOARD	Cut-and-paste data between applications (QClipboard)	QT_NO_MIME
	Sound	
QT_NO_SOUND	Playing audio files (QSound)	
	Scipting	
QT_NO_PROPERTIES	Scripting of Qt-based applications.	
	Qt/Embedded-specific	
QT_NO_QWS_CURSOR	The cursor sprite on Qt/Embedded. Pen-operated devices would not normally need this feature.	
QT_NO_QWS_DEPTH_8GRAYSCALE	8 bits per pixel: 256 levels of gray. Incompatible with QWS_DEPTH_8.	
QT_NO_QWS_DEPTH_8	8 bits per pixel: 216-color cube with 40 auxiliary colors. Incompatible with QWS_DEPTH_8GRAYSCALE.	
QT_NO_QWS_DEPTH_15	15 bits per pixel: 32 levels for each of red, green and blue.	
QT_NO_QWS_DEPTH_16	16 bits per pixel: 64 levels of green, 32 levels for red and for blue.	
QT_NO_QWS_DEPTH_32	32 bits per pixel: 256 levels for each of red, green and blue.	
QT_NO_QWS_MACH64	Mach64 accelerated driver (demonstration only).	
QT_NO_QWS_VFB	Virtual framebuffer running on X11 (see reference documentation).	
	Networking	
QT_NO_NETWORKPROTOCOL	Abstract multi-protocol data retrieval, with local file retrieval included (QNetworkProtocol)	
QT_NO_NETWORKPROTOCOL_FTP	FTP-protocol data retrieval.	QT_NO_NETWORKPROTOCOL
QT_NO_NETWORKPROTOCOL_HTTP	HTTP-protocol data retrieval.	QT_NO_NETWORKPROTOCOL
	Painting/drawing	
QT_NO_COLORNAMES	Color names such as "red", used by some QColor constructors and by some HTML documents (QColor, QStyleSheet)	
QT_NO_TRANSFORMATIONS	Used by a number of classes in Qt. With this, rotation and scaling are possible. Without it, only co-ordinate translation (QWMatrix)	
QT_NO_PSPRINTER	PostScript printer support.	
QT_NO_PRINTER	Printer support (QPrinter)	QT_NO_PSPRINTER (Unix only)
QT_NO_PICTURE	Save Qt drawing commands to a files (QPicture)	
	Widgets	
QT_NO_WIDGETS	Disabling this disables all widgets except QWidget.	
QT_NO_TEXTVIEW	HTML document viewing (QTextView)	QT_NO_WIDGETS, QT_NO_RICHTEXT
QT_NO_TEXTBROWSER	HTML document browsing (QTextBrowser)	QT_NO_TEXTVIEW

Porting to Qt 3.x

This document describes porting applications from Qt 2.x to Qt 3.x.

If you haven't yet made the decision about porting, or are unsure about whether it is worth it, take a look at the key features offered by Qt 3.x.

The Qt 3.x series is not binary compatible with the 2.x series. This means programs compiled for Qt 2.x must be recompiled to work with Qt 3.x. Qt 3.x is also not completely *source* compatible with 2.x, however all points of incompatibility cause compiler errors or run-time messages (rather than mysterious results). Qt 3.x includes many additional features and discards obsolete functionality. Porting from Qt 2.x to Qt 3.x is straightforward, and once completed makes the considerable additional power and flexibility of Qt 3.x available for use in your applications.

To port code from Qt 2.x to Qt 3.x:

1. Briefly read the porting notes below to get an idea of what to expect.
2. Be sure your code compiles and runs well on all your target platforms with Qt 2.x.
3. Recompile with Qt 3.x. For each error, search below for related identifiers (e.g. function names, class names). This document mentions all relevant identifiers to help you get the information you need at the cost of being a little verbose.
4. If you get stuck, ask on the qt-interest mailing list, or Trolltech Technical Support if you're a registered licensee.

Table of contents:

Header file inclusion changes

Qt 3.x remove some unnecessary nested #include directives from header files. This speeds up compilation when you don't need those nested header files. But in some cases you will find you need to add an extra #include to your files.

For example, if you get a message about QStringList or its functions not being defined, then add `#include <qstringlist.h>` at the top of the file giving the error.

Header files that you might need to add #include directives for include:

- `<qcursor.h>`
- `<qpainter.h>`
- `<qpen.h>`
- `<qstringlist.h>`
- `<qregexp.h>`
- `<qstrlist.h>`
- `<qstyle.h>`
- `<qvaluelist.h>`

Namespace

Qt 3.x is namespace clean. A few global identifiers that had been left in Qt 2.x have been discarded.

Enumeration Qt::CursorShape and its values are now part of the special Qt class defined in qnamespace.h. If you get compilation errors about these being missing (unlikely, since most of your code will be in classes that inherit from the Qt namespace class), then apply the following changes:

- `QCursorShape` becomes Qt::CursorShape
- `ArrowCursor` becomes Qt::ArrowCursor
- `UpArrowCursor` becomes Qt::UpArrowCursor
- `CrossCursor` becomes Qt::CrossCursor
- `WaitCursor` becomes Qt::WaitCursor
- `IbeamCursor` becomes Qt::IbeamCursor
- `SizeVerCursor` becomes Qt::SizeVerCursor
- `SizeHorCursor` becomes Qt::SizeHorCursor
- `SizeBDiagCursor` becomes Qt::SizeBDiagCursor
- `SizeFDiagCursor` becomes Qt::SizeFDiagCursor
- `SizeAllCursor` becomes Qt::SizeAllCursor
- `BlankCursor` becomes Qt::BlankCursor
- `SplitVCursor` becomes Qt::SplitVCursor
- `SplitHCursor` becomes Qt::SplitHCursor
- `PointingHandCursor` becomes Qt::PointingHandCursor
- `BitmapCursor` becomes Qt::BitmapCursor

The names of some debugging macro variables have been changed. We have tried not to break source compatibility as much as possible. If you observe error messages on the UNIX console or the Windows debugging stream that were previously disabled, please check these macro variables:

- `DEBUG` becomes `QT_DEBUG`
- `NO_DEBUG` becomes `QT_NO_DEBUG`
- `NO_CHECK` becomes `QT_NO_CHECK`
- `CHECK_STATE` becomes `QT_CHECK_STATE`
- `CHECK_RANGE` becomes `QT_CHECK_RANGE`
- `CHECK_NULL` becomes `QT_CHECK_NULL`
- `CHECK_MATH` becomes `QT_CHECK_MATH`

The name of some debugging macro functions has been changed as well but source compatibility should not be affected if the macro variable `QT_CLEAN_NAMESPACE` is not defined:

- `ASSERT` becomes `Q_ASSERT`
- `CHECK_PTR` becomes `Q_CHECK_PTR`

For the record, undocumented macro variables that are not part of the API have been changed:

- `_OS_*_` becomes `Q_OS_*`
- `_WS_*_` becomes `Q_WS_*`
- `_CC_*_` becomes `Q_CC_*`

Removed Functions

All these functions have been removed in Qt 3.x:

- QFont::charSet()
- QFont::setCharSet()
- QMenuBar::setActItem()
- QMenuBar::setWindowsAltMode()
- QPainter::drawQuadBezier()
- QPointArray::quadBezier()
- QPushButton::downButton()
- QPushButton::upButton()
- QRegExp::find()
- QSpinBox::downButton()
- QSpinBox::upButton()
- QString::basicDirection()
- QString::visual()
- QStyle::set...() functions
- QWidget::setFontPropagation()
- QWidget::setPalettePropagation()

Also, to avoid conflicts with `<iostream>`, the following three global functions have been renamed:

- setw() (renamed qSetW())
- setfill() (renamed qSetFill())
- setprecision() (renamed qSetPrecision())

Obsoleted Functions

The following functions have been obsoleted in Qt 3.0. The documentation of each of these functions should explain how to replace them in Qt 3.0.

Warning: It is best to consult http://doc.trolltech.com/3.0/ rather than the documentation supplied with Qt to obtain the latest information regarding obsolete functions and how to replace them in new code.

- QAccel::keyToString(QKeySequence k)
- QAccel::stringToKey(const QString & s)
- QActionGroup::insert(QAction *a)
- QButton::autoResize() const
- QButton::setAutoResize(bool)
- QCanvasItem::active() const
- QCanvasItem::enabled() const
- QCanvasItem::selected() const
- QCanvasItem::visible() const
- QCanvasPixmapArray::QCanvasPixmapArray(QPtrList<QPixmap> list, QPtrList<QPoint> hotspots)
- QCanvasPixmapArray::operator!()

- QColorGroup::QColorGroup(const QColor & foreground, const QColor & background, const QColor & light, const QColor & dark, const QColor & mid, const QColor & text, const QColor & base)
- QComboBox::autoResize() const
- QComboBox::setAutoResize(bool)
- QDate::dayName(int weekday)
- QDate::monthName(int month)
- QDir::encodedEntryList(const QString & nameFilter, int filterSpec = DefaultFilter, int sortSpec = DefaultSort) const
- QDir::encodedEntryList(int filterSpec = DefaultFilter, int sortSpec = DefaultSort) const
- QDockWindow::isHorizontalStretchable() const
- QDockWindow::isVerticalStretchable() const
- QDockWindow::setHorizontalStretchable(bool b)
- QDockWindow::setVerticalStretchable(bool b)
- QFont::defaultFont()
- QFont::setDefaultFont(const QFont & f)
- QFont::setPixelSizeFloat(float pixelSize)
- QFontDatabase::bold(const QString & family, const QString & style, const QString &) const
- QFontDatabase::families(bool) const
- QFontDatabase::font(const QString & familyName, const QString & style, int pointSize, const QString &)
- QFontDatabase::isBitmapScalable(const QString & family, const QString & style, const QString &) const
- QFontDatabase::isFixedPitch(const QString & family, const QString & style, const QString &) const
- QFontDatabase::isScalable(const QString & family, const QString & style, const QString &) const
- QFontDatabase::isSmoothlyScalable(const QString & family, const QString & style, const QString &) const
- QFontDatabase::italic(const QString & family, const QString & style, const QString &) const
- QFontDatabase::pointSizes(const QString & family, const QString & style, const QString &)
- QFontDatabase::smoothSizes(const QString & family, const QString & style, const QString &)
- QFontDatabase::styles(const QString & family, const QString &) const
- QFontDatabase::weight(const QString & family, const QString & style, const QString &) const
- QLabel::autoResize() const
- QLabel::setAutoResize(bool enable)
- QLineEdit::cursorLeft(bool mark, int steps = 1)
- QLineEdit::cursorRight(bool mark, int steps = 1)
- QLineEdit::hasMarkedText() const
- QLineEdit::markedText() const
- QLineEdit::repaintArea(int, int)
- QListBox::cellHeight(int i) const
- QListBox::cellHeight() const
- QListBox::cellWidth() const
- QListBox::findItem(int yPos) const
- QListBox::inSort(const QListBoxItem *lbi)
- QListBox::inSort(const QString & text)
- QListBox::itemYPos(int index, int *yPos) const
- QListBox::numCols() const
- QListBox::totalHeight() const

- QListBox::totalWidth() const
- QListBoxItem::current() const
- QListBoxItem::selected() const
- QListView::removeItem(QListViewItem *item)
- QListViewItem::removeItem(QListViewItem *item)
- QMainWindow::addToolBar(QDockWindow *, Dock = DockTop, bool newLine = FALSE)
- QMainWindow::addToolBar(QDockWindow *, const QString & label, Dock = DockTop, bool newLine = FALSE)
- QMainWindow::lineUpToolBars(bool keepNewLines = FALSE)
- QMainWindow::moveToolBar(QDockWindow *, Dock = DockTop)
- QMainWindow::moveToolBar(QDockWindow *, Dock, bool nl, int index, int extraOffset = -1)
- QMainWindow::removeToolBar(QDockWindow *)
- QMainWindow::setToolBarsMovable(bool)
- QMainWindow::toolBarPositionChanged(QToolBar *)
- QMainWindow::toolBarsMovable() const
- QMessageBox::message(const QString & caption, const QString & text, const QString & buttonText = QString::null, QWidget *parent = 0, const char *= 0)
- QMessageBox::query(const QString & caption, const QString & text, const QString & yesButtonText = QString::null, const QString & noButtonText = QString::null, QWidget *parent = 0, const char *= 0)
- QMessageBox::standardIcon(Icon icon, GUIStyle style)
- QRegExp::match(const QString & str, int index = 0, int *len = 0, bool indexIsStart = TRUE) const
- QScrollView::childIsVisible(QWidget *child)
- QScrollView::showChild(QWidget *child, bool show = TRUE)
- QSignal::block(bool b)
- QSignal::isBlocked() const
- QSignal::parameter() const
- QSignal::setParameter(int value)
- QSimpleRichText::draw(QPainter *p, int x, int y, const QRegion & clipRegion, const QColorGroup & cg, const QBrush *paper = 0) const
- QString::ascii() const
- QString::data() const
- QString::setExpand(uint index, QChar c)
- QStyle::defaultFrameWidth() const
- QStyle::scrollBarExtent() const
- QStyle::tabbarMetrics(const QWidget *t, int & hf, int & vf, int & ov) const
- QTabDialog::isTabEnabled(const char *name) const
- QTabDialog::selected(const QString &)
- QTabDialog::selected(const QString & tabLabel)
- QTabDialog::setTabEnabled(const char *name, bool enable)
- QTextStream::QTextStream(QString & str, int filemode)
- QToolBar::QToolBar(const QString & label, QMainWindow *, ToolBarDock = DockTop, bool newLine = FALSE, const char *name = 0)
- QToolButton::iconSet(bool on) const
- QToolButton::offIconSet() const
- QToolButton::onIconSet() const

- QToolButton::setIconSet(const QIconSet & set, bool on)
- QToolButton::setOffIconSet(const QIconSet &)
- QToolButton::setOnIconSet(const QIconSet &)
- QToolTip::enabled()
- QToolTip::setEnabled(bool enable)
- QTranslator::find(const char *context, const char *sourceText, const char *comment = 0) const
- QTranslator::insert(const char *context, const char *sourceText, const QString & translation)
- QTranslator::remove(const char *context, const char *sourceText)
- QUriDrag::setFilenames(const QStringList & fnames)
- QWidget::backgroundColor() const
- QWidget::backgroundPixmap() const
- QWidget::iconify()
- QWidget::setBackgroundColor(const QColor & c)
- QWidget::setBackgroundPixmap(const QPixmap & pm)
- QWidget::setFont(const QFont & f, bool)
- QWidget::setPalette(const QPalette & p, bool)
- QWizard::setFinish(QWidget *, bool)
- QXmlInputSource::QXmlInputSource(QFile & file)
- QXmlInputSource::QXmlInputSource(QTextStream & stream)
- QXmlReader::parse(const QXmlInputSource & input)

Additionally, these preprocessor directives have been removed:

- `#define strlen qstrlen`
- `#define strcpy qstrcpy`
- `#define strcmp qstrcmp`
- `#define strncmp qstrncmp`
- `#define stricmp qstricmp`
- `#define strnicmp qstrnicmp`

See the changes-3.0.0 document for an explanation of why this had to be done. You might have been relying on the non-portable and unpredictable behavior resulting from these directives. We strongly recommend that you either make use of the safe qstr* variants directly or ensure that no 0 pointer is passed to the standard C functions in your code base.

Collection Class Renaming

The classes QArray, QCollection, QList, QListIterator, QQueue, QStack and QVector have been renamed. To ease porting, the old names and the old header-file names are still supported.

Old Name	New Name	New Header File
QArray	QMemArray	<qmemarray.h>
QCollection	QPtrCollection	<qptrcollection.h>
QList	QPtrList	<qptrlist.h>
QListIterator	QPtrListIterator	<qptrlist.h>
QQueue	QPtrQueue	<qptrqueue.h>
QStack	QPtrStack	<qptrstack.h>
QVector	QPtrVector	<qptrvector.h>

QApplication

The function QApplication::processOneEvent() has been removed from the public API. Use QApplication::processEvents() instead.

QButtonGroup

In Qt 2.x, the function QButtonGroup::selected() returns the selected *radio* button (QRadioButton). In Qt 3.0, it returns the selected *toggle* button (QButton::toggleButton), a more general concept. This might affect programs that use QButtonGroups that contain a mixture of radio buttons and non-radio (e.g. QCheckBox) toggle buttons.

QDate

Two QDate member functions that were virtual in Qt 2.0 are not virtual in Qt 3.0. This is only relevant if you subclassed QDate and reimplemented these functions:

- QString QDate::monthName(int month) const
- QString QDate::dayName(int weekday) const

In addition to no longer being virtual, QDate::monthName() and QDate::dayName() have been renamed QDate::shortMonthN and QDate::shortDayName() and have been made static (as they should had been in the first place). The old names are still provided for source compatibility.

QFont

The internals of QFont have changed significantly between Qt 2.2 and Qt 3.0, to give better Unicode support and to make developing internationalized applications easier. The original API has been preserved with minimal changes. The CharSet enum and its related functions have disappeared. This is because Qt now handles all charset related issues internally, and removes this burden from the developer.

If you used the CharSet enum or its related functions, e.g QFont::charSet() or QFont::setCharSet(), just remove them from your code. There are a few functions that took a QFont::CharSet as a parameter; in these cases simply remove the charset from the parameter list.

QInputDialog

The two static getText(...) methods in QInputDialog have been merged. The `echo` parameter is the third parameter and defaults to QLineEdit::Normal.

If you used calls to QInputDialog::getText(...) that provided more than the first two required parameters you will must add a value for the `echo` parameter.

QLayout and Other Abstract Layout Classes

The definitions of QGLayoutIterator, QLayout, QLayoutItem, QLayoutIterator, QSpacerItem and QWidgetItem have been moved from `<qabstractlayout.h>` to `<qlayout.h>`. The header `<qabstractlayout.h>` now includes `<qlayout.h>` for compatibility. It might be removed in a future version.

QMultiLineEdit

The QMultiLineEdit was a simple editor widget in previous Qt versions. Since Qt 3.0 includes a new richtext engine, which also supports editing, QMultiLineEdit is obsolete. For the sake of compatibility QMultiLineEdit is still provided. It is now a subclass of QTextEdit which wraps the old QMultiLineEdit so that it is mostly source compatible to keep old applications working.

For new applications and when maintaining existing applications we recommend that you use QTextEdit instead of QMultiLineEdit wherever possible.

Although most of the old QMultiLineEdit API is still available, there is one important difference. The old QMultiLineEdit operated in terms of lines, whereas QTextEdit operates in terms of paragraphs. This is because lines change all the time during wordwrap, whereas paragraphs remain paragraphs. The consequence of this change is that functions which previously operated on lines, e.g. numLines(), textLine(), etc., now work on paragraphs.

Also the function getString() has been removed since it published the internal data structure.

In most cases, applications that used QMultiLineEdit will continue to work without problems. Applications that worked in terms of lines may require some porting.

The source code for the old 2.x version of QMultiLineEdit can be found in `$QTDIR/src/attic/qtmultilineedit.h/cpp`. Note that the class has been renamed to QtMultiLineEdit to avoid name clashes. If you really need to keep compatibility with the old QMultiLineEdit, simply include this class in your project and rename QMultiLineEdit to QtMultiLineEdit throughout.

QPrinter

QPrinter has undergone some changes, to make it more flexible and to ensure it has the same runtime behaviour on both Unix and Windows. In 2.x, QPrinter behaved differently on Windows and Unix, when using view transformations on the QPainter. This has changed now, and QPrinter behaves consistently across all platforms. A compatibilty mode has been added that forces the old behaviour, to ease porting from Qt 2.x to Qt 3.x. This compatibilty mode can be enabled by passing the QPrinter::Compatible flag to the QPrinter constructor.

On X11, QPrinter used to generate encapsulated postscript when fullPage() was TRUE and only one page was printed. This does not happen by default anymore, providing a more consistent printing output.

QRegExp

The QRegExp class has been rewritten to support many of the features of Perl regular expressions. Both the regular expression syntax and the QRegExp interface have been modified.

Be also aware that <qregexp.h> is no longer included automatically when you include <qstringlist.h>. See above for details.

New special characters

There are five new special characters: (,), {, | and } (parentheses, braces and pipe). When porting old regular expressions, you must add \ (backslash) in front of any of these (actually, \\ in C++ strings), unless it is already there.

Example: Old code like

```
QRegExp rx( "([0-9|]*\\)" );        // works in Qt 2.x
```

should be converted into

```
QRegExp rx( "\\([0-9\\|]*\\)" );      // works in Qt 2.x and 3.x
```

(Within character classes, the backslash is not necessary in front of certain characters, e.g. |, but it doesn't hurt.)

Wildcard patterns need no conversion. Here are two examples:

```
QRegExp wild( "(*.*)" );
wild.setWildcard( TRUE );

// TRUE as third argument means wildcard
QRegExp wild( "(*.*)", FALSE, TRUE );
```

However, when they are used, make sure to use QRegExp::exactMatch() rather than the obsolete QRegExp::match(). QRegExp::match(), like QRegExp::find(), tries to find a match somewhere in the target string, while QRegExp::exactMatch() tries to match the whole target string.

QRegExp::operator=()

This function has been replaced by QRegExp::setPattern() in Qt 2.2. Old code such as

```
QRegExp rx( "alpha" );
rx.setCaseSensitive( FALSE );
rx.setWildcard( TRUE );
rx = "beta";
```

still compiles with Qt 3, but produces a different result (the case sensitivity and wildcard options are forgotten). This way,

```
rx = "beta";
```

is the same as

```
rx = QRegExp( "beta" );
```

which is what one expects.

QRegExp::match()

The following function is now obsolete, as it has an unwieldy parameter list and was poorly named:

- bool QRegExp::match(const QString & str, int index = 0, int * len = 0, bool indexIsStart = TRUE) const

It will be removed in a future version of Qt. Its documentation explains how to replace it.

QRegExp::find()

This function was removed, after a brief appearance in Qt 2.2. Its name clashed with QString::find(). Use QRegExp::search() or QString::find() instead.

QString::findRev() and QString::contains()

QString::findRev()'s and QString::contains()'s semantics have changed between 2.0 and 3.0 to be more consistent with the other overloads.

For example,

```
QString( "" ).contains( QRegExp("") )
```

returns 1 in Qt 2.0; it returns 0 in Qt 3.0. Also, "^" now really means start of input, so

```
QString( "Heisan Hoppsan" ).contains( QRegExp("^.*$") )
```

returns 1, not 13 or 14.

This change affect very few existing programs.

QString::replace()

With Qt 1.0 and 2.0, a QString is converted implicitly into a QRegExp as the first argument to QString::replace():

```
QString text = fetch_it_from_somewhere();
text.replace( QString("[A-Z]+"), "" );
```

With Qt 3.0, the compiler gives an error. The solution is to use a QRegExp cast:

```
text.replace( QRegExp("[A-Z]+"), "" );
```

This change makes it possible to introduce a QString::replace(QString, QString) overload in a future version of Qt without breaking source compatibility.

QSortedList

The QSortedList class is now obsolete. Consider using a QDict, a QMap or a plain QPtrList instead.

QTableView

The QTableView class has been obsoleted and is no longer a part of the Qt API. Either use the powerful QTable class or the simplistic QGridView in any new code you create. If you really need the old table view for compatibility you can find it in `$QTDIR/src/attic/qttableview.{cpp,h}`. Note that the class has been renamed from QTableView to QtTableView to avoid name clashes. To use it, simply include it in your project and rename QTableView to QtTableView throughout.

QToolButton

The QToolButton class used to distinguish between "on" and "off" icons. In 3.0, this mechanism was moved into the QIconSet class (see QIconSet::State).

The old QToolButton::onIconSet and QToolButton::offIconSet properties are still provided so that old source will compile, but their semantics have changed: they are now synonyms for QToolButton::iconSet. If you used that distinction in Qt 2.x, you will need to adjust your code to use the QIconSet On/Off mechanism.

Likewise, the *on* parameter of these two functions is now ignored:

- void QToolButton::setIconSet (const QIconSet & set, bool on)
- QIconSet QToolButton::iconSet (bool on) const

These functions are only provided for ease of porting. New code should use the following instead:

- void QToolButton::setIconSet(const QIconSet & set)
- QIconSet QToolButton::iconSet() const

Finally, this function is no longer virtual:

- void QToolButton::setIconSet(const QIconSet & set, bool on)

If you have a class that inherits QToolButton and that reimplements QToolButton::setIconSet(), you should make the signature of the reimplementation agree with the new QToolButton::setIconSet(), a virtual function.

QTextStream

The global QTextStream manipulators setw(), setfill() and setprecison() were renamed to qSetW(), qSetFill() and qSetPrecision() to avoid conflicts with `<iostream.h>`. If you used them, you must rename the occurrences to the new names.

QTranslator

The QTranslator class was extended in Qt 2.2, and these extensions lead to a new interface. This interface is used mainly by translation tools (for example, Qt Linguist). For source compatibility, no member function was effectively removed. The QTranslator documentation points out which functions are obsolete.

This function is no longer virtual:

- QString QTranslator::find(const char * context, const char * sourceText) const

If you have a class that inherits QTranslator and which reimplements QTranslator::find(), you should reimplement QTranslator::findMessage() instead. In fact, find() is now defined in terms of findMessage(). By doing the conversion, you will also gain support for translator comments and for any future extensions.

QWidget

QWidget::backgroundColor(), QWidget::setBackgroundColor(), QWidget::backgroundPixmap() and QWidget::setBackgroun have often been the source of much confusion in previous releases. Qt 3.0 addresses this by obsoleting these functions and by remplacing them with eight new functions: QWidget::eraseColor(), QWidget::setEraseColor(), QWidget::erasePixmap(), QWidget::setErasePixmap(), QWidget::paletteBackgroundColor(), QWidget::setPaletteBackgroundColor QWidget::paletteBackgroundPixmap() and QWidget::setPaletteBackgroundPixmap(). See their documentation for details.

QXml Classes

QXmlInputSource

The semantics of QXmlInputSource has changed slightly. This change only affects code that parses the same data from the same input source multiple times. In such cases you must call QXmlInputSource::reset() before the second

call to QXmlSimpleReader::parse().

So code like

```
QXmlInputSource source( &xmlFile );
QXmlSimpleReader reader;
...
reader.parse( source );
...
reader.parse( source );
```

must be changed to

```
QXmlInputSource source( &xmlFile );
QXmlSimpleReader reader;
...
reader.parse( source );
...
source.reset();
reader.parse( source );
```

QXmlLocator

Due to some internal changes, it was necessary to clean-up the semantics of QXmlLocator: this class is now an abstract class. This shouldn't cause any problems, since programmers usually used the QXmlLocator that was reported by QXmlContentHandler::setDocumentLocator(). If you used this class in some other way, you must adjust your code to use the QXmlLocator that is reported by the QXmlContentHandler::setDocumentLocator() function.

Asynchronous I/O Classes

QASyncIO, QDataSink, QDataSource, QIODeviceSource and QDataPump were used internally in previous versions of Qt, but are not used anymore. They are now obsolete.

Bezier Curves

The function names for Bezier curves in QPainter and QPointArray have been corrected. They now properly reflect their cubic form instead of a quadratic one. If you have been using either QPainter::drawQuadBezier() or QPointArray::quadBezier() you must replace these calls with

- void QPainter::drawCubicBezier(const QPointArray &, int index=0) and
- QPointArray QPointArray::cubicBezier() const

respectively. Neither the arguments nor the resulting curve have changed.

Locale-aware String Comparisons in QIconView, QListBox, QListView and QTable

In Qt 2.x, QString only provided string comparisons using the Unicode values of the characters of a string. This is efficient and reliable, but it is not the appropriate order for most languages. For example, French users expect 'é' (e acute) to be treated essentially as 'e' and not put after 'z'.

In Qt 3.0, QString::localeAwareCompare() implements locale aware string comparisions on certain platforms. The classes QIconView, QListBox, QListView and QTable now use QString::localeAwareCompare() instead of QString::compare(). If you want to control the behaviour yourself you can always reimplement QIconViewItem::compare(), QListBox::text(), QListViewItem::compare() or QTableItem::key() as appropriate.

Session Management

Definitions

A *session* is a group of applications running, each of which has a particular state. The session is controlled by a service called the *session manager*. The applications participating in the session are called *session clients*.

The session manager issues commands to its clients on behalf of the user. These commands may cause clients to commit unsaved changes (for example by saving open files), to preserve their state for future sessions or to terminate gracefully. The set of these operations is called *session management*.

In the common case, a session consists of all applications that a user runs on their desktop at a time. Under Unix/X11, however, a session may include applications running on different computers and may span multiple displays.

Shutting a session down

A session is shut down by the session manager, usually on behalf of the user when they want to log out. A system might also perform an automatic shutdown in an emergency situation, for example, if power is about to be lost. Clearly there is a significant big difference between both shutdowns. During the first, the user may want to interact with the application, specifying exactly which files should be saved and which should be discarded. In the latter case, there's no time for interaction. There may not even be a user sitting in front of the machine!

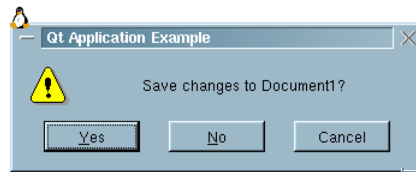
Protocols and support on different platforms

On Mac OS X and MS-Windows, there is nothing like complete session management for applications yet, i.e. no restoring of previous sessions. They do support graceful logouts where applications have the chance to cancel the process after getting confirmation from the user. This is the functionality that corresponds to the QApplication::commitData() method.

X11 has supported complete session management since X11R6.

Getting session management to work with Qt

Start by reimplementing QApplication::commitData() to enable your application to take part in the graceful logout process. If you target the MS-Windows platform only, this is all you can and have to provide. Ideally, your application should provide a shutdown dialog similar to the following one:



Example code to this dialog can be found in the documentation of QSessionManager::allowsInteraction().

For complete session management (only supported on X11R6 at present), you also have to take care of saving the state of the application and potentially restore the state in the next life cycle of the session. This saving is done by reimplementing QApplication::saveState(). All state data you are saving in this function, should be marked with the session identifier QApplication::sessionId(). This application specific identifier is globally unique, so no clashes will occur. (See QSessionManager for information on saving/restoring the state of a particular Qt application.)

Restoration is usually done in the application's main() function. Check if QApplication::isSessionRestored() is `TRUE`. If that's the case, use the session identifier QApplication::sessionId() again to access your state date and restore the state of the application.

Important: In order to allow the window manager to restore window attributes such as stacking order or geometry information, you must identify your top level widgets with unique application-wide object names (see QObject::setName()). When restoring the application, you must ensure that all restored top level widgets are given the same unique names they had before.

Testing and debugging session management

Session management support on Mac OS X and Windows is fairly limited due to the lack of this functionality in the operating system itself. Simply shut the session down and verify that your application behaves as wanted. It may be a good idea to launch another application, usually the integrated development environment, before starting your application. This other application will get the shutdown message afterwards, thus permitting you to cancel the shutdown. Otherwise you would have to log in again after each test run, which is not a problem per se but time consuming.

On Unix you can either use a desktop environment that supports standard X11R6 session management or, the recommended method, use the session manager reference implementation provided by the X Consortium. This sample manager is called `xsm` and is part of a standard X11R6 installation. As always with X11, a useful and informative manual page is provided. Using `xsm` is straightforward (apart from the clumsy Athena-based user interface). Here's a simple approach:

- Run X11R6.
- Create a dot file `.xsmstartup` in your home directory which contains the single line

  ```
  xterm
  ```

  This tells `xsm` that the default/failsafe session is just an xterm and nothing else. Otherwise `xsm` would try to invoke lots of clients including the windowmanager `twm`, which isn't very helpful.
- Now launch `xsm` from another terminal window. Both a session manager window and the xterm will appear. The xterm has a nice property that sets it apart from all the other shells you are currently running: within its shell, the `SESSION_MANAGER` environment variable points to the session manager you just started.
- Launch your application from the new xterm window. It will connect itself automatically to the session manager. You can check with the *ClientList* push button whether the connect was successful. Note: Never keep the *ClientList* open when you start or end session managed clients! Otherwise `xsm` is likely to crash.
- Use the session manager's *Checkpoint* and *Shutdown* buttons with different settings and see how your application behaves. The save type *local* means that the clients should save their state. It corresponds to the QApplication::saveState() function. The *global* save type asks application to save their unsaved changes in the permanent, globally accessible storage. It invokes QApplication::commitData().

- Whenever something crashes, blame `xsm` and not Qt. `xsm` is far from being a usable session manager on a user's desktop. It is, however, stable and useful enough to serve as testing environment.

Shared Classes

Many C++ classes in Qt use *explicit* and *implicit* data sharing to maximize resource usage and minimize copying of data.

Overview

A shared class consists of a pointer to a shared data block that contains a reference count and the data.

When a shared object is created, it sets the reference count to 1. The reference count is incremented whenever a new object references the shared data, and decremented when the object dereferences the shared data. The shared data is deleted when the reference count becomes zero.

When dealing with shared objects, there are two ways of copying an object. We usually speak about *deep* and *shallow* copies. A deep copy implies duplicating an object. A shallow copy is a reference copy, we only copy a pointer to a shared data block. Making a deep copy can be expensive in terms of memory and CPU. Making a shallow copy is very fast, because it only involves setting a pointer and incrementing the reference count.

Object assignment (with operator=) for implicitly and explicitly shared objects is implemented as shallow copies. A deep copy can be made by calling a copy() function.

The benefit of sharing is that a program does not need to duplicate data when it is not required, which results in less memory usage and less copying of data. Objects can easily be assigned, sent as function arguments and returned from functions.

Now comes the distinction between *explicit* and *implicit* sharing. Explicit sharing means that the programmer must be aware of the fact that objects share common data. Implicit sharing means that the sharing mechanism takes place behind the scenes and the programmer does not need to worry about it.

A QByteArray Example

QByteArray is an example of a shared class that uses explicit sharing. Example:

```
                        //        a=         b=         c=
QByteArray a(3),b(2)    // 1)     {?,?,?}    {?,?}
b[0] = 12; b[1] = 34;   // 2)     {?,?,?}    {12,34}
a = b;                  // 3)     {12,34}    {12,34}
a[1] = 56;              // 4)     {12,56}    {12,56}
QByteArray c = a;       // 5)     {12,56}    {12,56}    {12,56}
a.detach();             // 6)     {12,56}    {12,56}    {12,56}
a[1] = 78;              // 7)     {12,78}    {12,56}    {12,56}
b = a.copy();           // 8)     {12,78}    {12,78}    {12,56}
a[1] = 90;              // 9)     {12,90}    {12,78}    {12,56}
```

The assignment `a = b` on line 3 throws away `a`'s original shared block (the reference count becomes zero), sets `a`'s shared block to point to `b`'s shared block and increments the reference count.

On line 4, the contents of `a` is modified. `b` is also modified, because `a` and `b` refer the same data block. This is the difference between explicit and implicit sharing (explained below).

The `a` object detaches from the common data on line 6. Detaching means to copy the shared data to make sure that an object has its own private data. Therefore, modifying `a` on line 7 will not affect `b` or `c`.

Finally, on line 8 we make a deep copy of `a` and assign it to `b`, so that when `a` is modified on line 9, `b` remains unchanged.

Explicit vs. Implicit Sharing

Implicit sharing automatically detaches the object from a shared block if the object is about to change and the reference count is greater than one. Explicit sharing leaves this job to the programmer. If an explicitly shared object is not detached, changing the object will change all other objects that refer to the same data.

Implicit sharing optimizes memory usage and copying of data without this side effect. So why didn't we implement implicit sharing for all shared classes? The answer is that a class that allows direct access to its internal data (for efficiency reasons), like QByteArray, cannot be implicitly shared, because it can be changed without letting QByteArray know.

An implicitly shared class has total control of its internal data. In any member functions that modify its data, it automatically detaches before modifying the data.

The QPen class, which uses implicit sharing, detaches from the shared data in all member functions that change the internal data.

Code fragment:

```
void QPen::setStyle( PenStyle s )
{
    detach();        // detach from common data
    data->style = s; // set the style member
}

void QPen::detach()
{
    if ( data->count != 1 ) // only if >1 reference
        *this = copy();
}
```

This is clearly not possible for QByteArray, because the programmer can do the following:

```
QByteArray array( 10 );
array.fill( 'a' );
array[0] = 'f';        // will modify array
array.data()[1] = 'i'; // will modify array
```

If we monitor changes in a QByteArray, the QByteArray class would become unacceptably slow.

Explicitly Shared Classes

All classes that are instances of the QMemArray template class are explicitly shared:

- QBitArray
- QPointArray

- QByteArray
- Any other instantiation of QMemArray<type>

These classes have a detach() function that can be called if you want your object to get a private copy of the shared data. They also have a copy() function that returns a deep copy with a reference count of 1.

The same is true for QImage, which does not inherit QMemArray. QMovie is also explicitly shared, but it does not support detach() and copy().

Implicitly Shared Classes

The Qt classes that are implicitly shared are:

- QBitmap
- QBrush
- QCursor
- QFont
- QFontInfo
- QFontMetrics
- QIconSet
- QMap
- QPalette
- QPen
- QPicture
- QPixmap
- QRegion
- QRegExp
- QString
- QStringList
- QValueList
- QValueStack

These classes automatically detach from common data if an object is about to be changed. The programmer will not even notice that the objects are shared. Thus you should treat separate instances of them as separate objects. They will always behave as separate objects but with the added bonus of sharing data whenever possible. For this reason, you can pass instances of these classes as arguments to functions by value without concern for the copying overhead.

Example:

```
QPixmap p1, p2;
p1.load( "image.bmp" );
p2 = p1;                    // p1 and p2 share data
QPainter paint;
paint.begin( &p2 );         // cuts p2 loose from p1
paint.drawText( 0,50, "Hi" );
paint.end();
```

In this example, `p1` and `p2` share data until QPainter::begin() is called for `p2`, because painting a pixmap will modify it. The same happens also if anything is bitBlt()'ed into `p2`.

QCString: implicit or explicit?

QCString uses a mixture of implicit and explicit sharing. Functions inherited from QByteArray, such as data(), employ explicit sharing, while those only in QCString detach automatically. Thus, QCString is rather an "experts only" class, provided mainly to ease porting from Qt 1.x to Qt 2.0. We recommend that you use QString, a purely implicitly shared class.

Year 2000 Compliance Statement

Trolltech defines *Year 2000 Compliance* as a requirement that a product or part of product does not contain errors related to transition from December 31, 1999 to January 1, 2000, or to the existence of February 29, 2000.

This document certifies that the API provided by Qt and the implementation of Qt are both Year 2000 Compliant, and that the use of underlying APIs by Qt does not have any known problems.

The API Provided by Qt

Several parts of Qt deal with dates and times:

- QDate - provides date management
- QDateTime - provides date/time management
- QTime - provides time management (within a date)
- QTimer - provides delayed or regular execution of code.

All of these classes' external APIs are Year 2000 Compliant: QDate and QDateTime offer only four-digit years as output, QTime and QTimer do not deal with years or leap days at all.

Implementation Issues in Qt

All date/time calculation and storage in Qt uses number of days, seconds or milliseconds, and is thus Year 2000 Compliant.

This applies to the above four classes and also to QFileDialog (which can sort files by time/date), QFileInfo (which operates on file times/dates) and QApplication (which does various internal housekeeping tasks).

The conversion to `year/month/date` format in QDate (and QDateTime) has been verified to be correct for all of December 31, 1999, January 1, 2000, February 28 and 29, 2000, March 1, 2000, January 1, 2001 and March 1, 2001.

Qt has been verified to be robust in case of time/date errors (such as time warps) in the underlying operating system.

Use of System APIs

It is of course impossible for Trolltech to ensure that both of the window systems and all of the operating systems on which Qt runs are Year 2000 Compliant. However, Qt does not use any APIs that are known to have any Year 2000-related bugs, or seem at risk to have any.

Common Issues and Special Use Cases

This document describes how to use more than one Qt version on one machine and how to use Qt on X11 without a window manager. In addition it explains the most common source of link errors with Qt.

Other frequently asked questions can be found in the FAQ index .

Link error, complaining about a lack of vtbl, _vtbl, __vtbl or similar

This indicates that you've included the Q_OBJECT macro in a class declaration and probably also run the moc, but forgot to link the moc-generated object code into your executable. See Using the Meta Object Compiler for details on how to use moc.

Using different versions of Qt on the same machine

Qt programs need the following components of a Qt distribution:

Header files - Compile time

Programmers need to include the Qt header files. The Qt header files are usually located in the `include` subdirectory of Qt distributions. Care must be taken to include the header files of the relevant release of Qt. Those with a command-line compiler will typically use options such as `/I%QTDIR%\include`

the relevant release of Qt.

Meta Object Compiler and other tools - Compile time

Programmers need to run *moc* and other tools such as *uic*. These tools are usually located in the `bin` subdirectory of Qt distributions. Either run `"$QTDIR"/bin/moc` and `"$QTDIR"/bin/uic` or add `"$QTDIR"/bin` to your `PATH` and run moc and `uic`. If you use `qmake` the appropriate lines will be added to your Makefiles so that *uic* and *moc* will be executed as required.

Static or shared libraries - Link time

Programmers need to link with the Qt static or shared libraries. The Qt libraries are usually located in the `lib` subdirectory of Qt distributions. Care must be taken to link with the libraries of the relevant release of Qt. Those with a command-line compiler will typically use options such as `/L%QTDIR%\lib\qt.lib` or `-L"$QTDIR"/lib -lqt` provided `QTDIR` specifies the relevant release of Qt.

Shared libraries - Run time

Users of programs linked with shared Qt libraries need these same shared libraries to run these programs. The Qt libraries are usually located in the `lib` subdirectory of Qt distributions. Shared libraries are made available to programs in places such as `C:\windows\system` on Windows platforms, directories listed in file `/etc/ld.so.conf` on Linux, standard `lib` directories on Unix, or directories listed in environment variables `LD_LIBRARY_PATH`, `SHLIB_PATH`, or `LIBPATH` on various Unix flavours. Make the relevant Qt libraries available using one of these mechanisms.

Qt distributions consist of different files needed at compile time, link time, or run time. Trolltech distributes Qt in the form of a source package that contain all these files once they have been built.

Other vendors distribute Qt in the form of binary packages. Binary packages usually consist of two parts:

- shared libraries in the run time package, usually called `qt3`.
- header files, static libraries, the moc and other tools in the developers' kit, usually called `qt3-dev`.

Depending on how you are using Qt, you need to make specific parts of the Qt distribution available to your programs. Typical situations are described below.

Developers building for a single version of Qt on Unix - Qt binary packages

You build programs with a single version of Qt, but you still need to run programs linked with another version of Qt. You are typically a Linux developer who builds programs for Qt 3.x on a KDE desktop based on Qt 2.x. Qt packages are usually split into a shared library package with a name like `qt` and a developer package with a name like `qt-dev`. You will need the appropriate packages:

- To build programs you will need the header files, the libraries, the moc and other tools from Qt 3.x. They are included in the developer package of Qt 3.x (`qt3-dev` or similar).
- To run programs you will need the shared libraries of Qt 3.x and Qt 2.x. They are included in the regular packages of Qt 3.x (`qt3` or similar) and Qt 2.x (`qt2` or similar).

Just install the packages, `qt2`, `qt3`, and `qt3-dev`. You may need to set the environment variable `QTDIR` to point to Qt 3.x.

Developers building for two versions of Qt on Unix - Qt sources

You build and run programs for Qt 2.x and Qt 3.x. You will need:

- the header files, the libraries, the moc and other tools from Qt 3.x and Qt 2.x to build programs,
- the shared libraries of Qt 3.x and Qt 2.x to run programs.

Get the source distributions of both Qt 2.x and Qt 3.x.

1. Install and build Qt 2.x and Qt 3.x, usually in `/opt` or `/usr/local`. In the case of `/opt`:

   ```
   $ cd /opt
   $ gunzip -c \c qt-x11-2.3.1.tar.gz | tar xf -
   $ cd qt-2.3.1
   $ setenv QTDIR /opt/qt-2.3.1
   $ configure [options]
   $ make

   $ cd /opt
   $ gunzip -c qt-x11-free-3.0.0.tar.gz | tar xf -
   $ cd qt-3.0.0
   $ setenv QTDIR /opt/qt-3.0.0
   $ configure [options]
   $ make
   ```

2. Make shared libraries available to programs at run time. Either add both `/opt/qt-2.3.1/lib` and `/opt/qt-3.0.0/lib` to your environment variable `LD_LIBRARY_PATH` or file `/etc/ld.so.conf` or whatever mechanism you're using, or make links to the libraries in a standard directory like `/usr/local/lib`:

```
cd /usr/local/lib
ln -s /opt/qt-2.3.1/lib/libqt.so.2 .
ln -s /opt/qt-2.3.1/lib/libqt-mt.so.2 .
ln -s /opt/qt-2.3.1/lib/libqutil.so.1 .
ln -s /opt/qt-3.0.0/lib/libqt.so.3 .
ln -s /opt/qt-3.0.0/lib/libqui.so.1 .
```

To develop with Qt 2.x use:

```
setenv QTDIR /opt/qt-2.3.1
setenv PATH ${QTDIR}/bin:${PATH}
```

To develop with Qt 3.x use:

```
setenv QTDIR /opt/qt-3.0.0
setenv PATH ${QTDIR}/bin:${PATH}
```

Setting `QTDIR` ensures that the proper resources are used, such as the documentation appropriate to the version of Qt you're using. Also your Makfiles may refer to `"$QTDIR"/include` and `"$QTDIR"/lib` to include the proper header files and link with the proper libraries. Setting the `PATH` ensures that the proper version of moc and other tools is being used.

Using Qt on X11 without a window manager

When using Qt without a window manager on Unix/X11, you will most likely experience focus problems. Without a window manager, there is no focus handling on X11, and no concept of an active window either. If you want your application to work in such an environment, you have to explicitly mark a window as active *after* showing it:

```
yourWindow->show();
yourWindow->setActiveWindow();
```

Note that setActiveWindow() won't work if the widget does not become physically visible during this event cycle. However, without a window manager running, this is guaranteed to happen. For the curious reader: setActiveWindow() emulates a window manager by explicitly setting the X Input Focus to a widget's top level window.

Other common problems

Other common problems are covered by the online Technical FAQ.

How to Report A Bug

If you think you have found a bug in Qt, we would like to hear about it so we can fix it.

Before reporting a bug, please check the FAQs and Platform Notes on our web site to see if the issue is already known.

Always include the following information in your bug report:

1. The name and version number of your compiler
2. The name and version number of your operating system
3. The version of Qt you are using, and what configure options it was compiled with.

If the problem you are reporting is only at visible run-time, try to create a small test program that shows the problem when run. Often, such a program can be create with some minor changes to one of the many example programs in the `qt/examples` directory.

Please send the bug report to qt-bugs@trolltech.com.

Installing Qt/Windows

The Qt/Windows distribution is distributed as a self-extracting archive with a built-in installer. Just follow the installation wizard.

Installing Qt/X11

You may need to be root, depending on the permissions of the directories where you choose to install Qt.

1. Unpack the archive if you have not done so already:

   ```
   cd /usr/local
   gunzip qt-x11-version.tar.gz    # uncompress the archive
   tar xf qt-x11-version.tar       # unpack it
   ```

   This creates the directory /usr/local/qt-*version* containing the files from the main archive.

   Rename qt-*version* to qt (or make a symlink):

   ```
   mv qt-version qt
   ```

   The rest of this file assumes that Qt is installed in /usr/local/qt.

2. Set some environment variables in the file .profile (or .login, depending on your shell) in your home directory. Create the file if it is not there already.

   - QTDIR — wherever you installed Qt
   - PATH — to locate the moc program and other Qt tools
   - MANPATH — to access the Qt man pages
   - LD_LIBRARY_PATH — for the shared Qt library

   This is done like this:

   In .profile (if your shell is bash, ksh, zsh or sh), add the following lines:

   ```
   QTDIR=/usr/local/qt
   PATH=$QTDIR/bin:$PATH
   MANPATH=$QTDIR/man:$MANPATH
   LD_LIBRARY_PATH=$QTDIR/lib:$LD_LIBRARY_PATH

   export QTDIR PATH MANPATH LD_LIBRARY_PATH
   ```

   In .login (in case your shell is csh or tcsh), add the following lines:

   ```
   setenv QTDIR /usr/local/qt
   setenv PATH $QTDIR/bin:$PATH
   setenv MANPATH $QTDIR/man:$MANPATH
   setenv LD_LIBRARY_PATH $QTDIR/lib:$LD_LIBRARY_PATH
   ```

   After you have done this, you will need to login again, or re-source the profile before continuing, so that at least $QTDIR is set. The installation will give an error message and not proceed otherwise.

3. Install your license file. For the free edition, you do not need a license file. For Professional and Enterprise editions, install your license file as described in your distribution.
4. Compile the Qt library, and build the example programs, the tutorial and the tools (eg. the Designer) as follows.

   Type:

   ```
   ./configure
   ```

   This will configure the Qt library for your machine. Note that GIF support is turned off by default. Run ./configure -help to get a list of configuration options. Read PLATFORMS for a list of supported platforms.

   To create the library and compile all examples and the tutorial:

   ```
   make
   ```

   If your platform or compiler is not supported, read PORTING. If it is supported but you have problems, see http://www.trolltech.com/platforms/.
5. In very few cases you may need to run /sbin/ldconfig or something similar at this point if you are using shared libraries.

   If you have problems running the example programs, e.g. messages like

   ```
   can't load library 'libqt.so.2'
   ```

   you probably need to put a reference to the qt library in a configuration file and run /sbin/ldconfig as root on your system. And don't forget to set LD_LIBRARY_PATH as explained in 2) above.
6. The online HTML documentation is installed in /usr/local/qt/doc/html/ The main page is /usr/local/qt/doc/html/inde
   The man pages are installed in /usr/local/qt/doc/man/

You're done. Qt is now installed.

Qt Free Edition License Agreement

The Qt Free Edition is distributed under the Q Public License (QPL). It allows free use of Qt Free Edition for running software developed by others, and free use of Qt Free Edition for development of free/Open Source software. There is more information about the QPL at the Trolltech web site.

Note that the Qt/Embedded Free Edition is **not** distributed under the QPL, but under the GNU General Public License (GPL).

For development non-free/proprietary software, the Qt Professional Edition is available. It has a normal commercial library license, with none of the special restrictions of the QPL or the GPL.

THE Q PUBLIC LICENSE version 1.0

Copyright (C) 1999-2000 Trolltech AS, Norway. Everyone is permitted to copy and distribute this license document.

The intent of this license is to establish freedom to share and change the software regulated by this license under the open source model.

This license applies to any software containing a notice placed by the copyright holder saying that it may be distributed under the terms of the Q Public License version 1.0. Such software is herein referred to as the Software. This license covers modification and distribution of the Software, use of third-party application programs based on the Software, and development of free software which uses the Software.

Granted Rights

1. You are granted the non-exclusive rights set forth in this license provided you agree to and comply with any and all conditions in this license. Whole or partial distribution of the Software, or software items that link with the Software, in any form signifies acceptance of this license.

2. You may copy and distribute the Software in unmodified form provided that the entire package, including - but not restricted to - copyright, trademark notices and disclaimers, as released by the initial developer of the Software, is distributed.

3. You may make modifications to the Software and distribute your modifications, in a form that is separate from the Software, such as patches. The following restrictions apply to modifications:

a. Modifications must not alter or remove any copyright notices in the Software.

b. When modifications to the Software are released under this license, a non-exclusive royalty-free right is granted to the initial developer of the Software to distribute your modification in future versions of the Software provided such versions remain available under these terms in addition to any other license(s) of the initial developer.

4. You may distribute machine-executable forms of the Software or machine-executable forms of modified versions of the Software, provided that you meet these restrictions:

a. You must include this license document in the distribution.

b. You must ensure that all recipients of the machine-executable forms are also able to receive the complete machine-readable source code to the distributed Software, including all modifications, without any charge beyond the costs of data transfer, and place prominent notices in the distribution explaining this.

c. You must ensure that all modifications included in the machine-executable forms are available under the terms of this license.

5. You may use the original or modified versions of the Software to compile, link and run application programs legally developed by you or by others.

6. You may develop application programs, reusable components and other software items that link with the original or modified versions of the Software. These items, when distributed, are subject to the following requirements:

a. You must ensure that all recipients of machine-executable forms of these items are also able to receive and use the complete machine-readable source code to the items without any charge beyond the costs of data transfer.

b. You must explicitly license all recipients of your items to use and re-distribute original and modified versions of the items in both machine-executable and source code forms. The recipients must be able to do so without any charges whatsoever, and they must be able to re-distribute to anyone they choose.

c. If the items are not available to the general public, and the initial developer of the Software requests a copy of the items, then you must supply one.

Limitations of Liability

In no event shall the initial developers or copyright holders be liable for any damages whatsoever, including - but not restricted to - lost revenue or profits or other direct, indirect, special, incidental or consequential damages, even if they have been advised of the possibility of such damages, except to the extent invariable law, if any, provides otherwise.

No Warranty

The Software and this license document are provided AS IS with NO WARRANTY OF ANY KIND, INCLUDING THE WARRANTY OF DESIGN, MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Choice of Law

This license is governed by the Laws of Norway. Disputes shall be settled by Oslo City Court.

Licenses for Code Used in Qt

Qt contains a little code that is not under the QPL, the GPL, or the Qt Commercial License Agreement, but rather under specific highly permissive license from the original authors. This page lists the licenses used for that code, names the authors, and links to the points where it is used.

Trolltech gratefully acknowledges these and others contribution to Qt. We recommend that all programs that use Qt also acknowledge these contributions, and quote all these license statements in an appendix to the documentation.

Copyright (C) 1989, 1991 by Jef Poskanzer.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. This software is provided "as is" without express or implied warranty.

- QImage::smoothScale

Copyright (c) 1999 Mizi Research Inc. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. \sa

- QEucKrCodec

Copyright (c) 1999 Serika Kurusugawa. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS "AS IS". ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. \sa

- QEucJpCodec
- QJisCodec
- QSjisCodec

Copyright 1995, Trinity College Computing Center. Written by David Chappell.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. This software is provided "as is" without express or implied warranty.

TrueType font support. These functions allow PPR to generate PostScript fonts from Microsoft compatible TrueType font files.

The functions in this file do most of the work to convert a TrueType font to a type 3 PostScript font.

Most of the material in this file is derived from a program called "ttf2ps" which L. S. Ng posted to the usenet news group "comp.sources.postscript". The author did not provide a copyright notice or indicate any restrictions on use.

Last revised 11 July 1995.

\sa

- QPrinter

Copyright 1996 Daniel Dardailler.

Permission to use, copy, modify, distribute, and sell this software for any purpose is hereby granted without fee, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Daniel Dardailler not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. Daniel Dardailler makes no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

Modifications Copyright 1999 Matt Koss, under the same license as above.

- Drag and Drop

Copyright 2000 Hans Petter Bieker . All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY

AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. \sa

- QTsciiCodec

Copyright 2000 TurboLinux, Inc. Written by Justin Yu and Sean Chen.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS "AS IS", AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. \sa

- QGbkCodec

Qt supports GIF reading if it is configured that way during installation (see qgif.h). If it is, we are required to state that "The Graphics Interchange Format(c) is the Copyright property of CompuServe Incorporated. GIF(sm) is a Service Mark property of CompuServe Incorporated." \sa

- QImageDecoder

GNU General Public License

The Qt GUI Toolkit is Copyright (C) 1994-2001 Trolltech AS.

The Qt Free Edition and the Qt/Embedded Free Edition are available under the GPL. The Qt Free Edition (for Unix/X11) is also available under the QPL.

```
 You may use, distribute and copy the Qt GUI Toolkit under the terms of
 GNU General Public License version 2, which is displayed below.

                    GNU GENERAL PUBLIC LICENSE
                       Version 2, June 1991

 Copyright (C) 1989, 1991 Free Software Foundation, Inc.
                          675 Mass Ave, Cambridge, MA 02139, USA
 Everyone is permitted to copy and distribute verbatim copies
 of this license document, but changing it is not allowed.

                            Preamble

  The licenses for most software are designed to take away your
freedom to share and change it.  By contrast, the GNU General Public
License is intended to guarantee your freedom to share and change free
software--to make sure the software is free for all its users.  This
General Public License applies to most of the Free Software
Foundation's software and to any other program whose authors commit to
using it.  (Some other Free Software Foundation software is covered by
the GNU Library General Public License instead.)  You can apply it to
your programs, too.

  When we speak of free software, we are referring to freedom, not
price.  Our General Public Licenses are designed to make sure that you
have the freedom to distribute copies of free software (and charge for
this service if you wish), that you receive source code or can get it
if you want it, that you can change the software or use pieces of it
in new free programs; and that you know you can do these things.

  To protect your rights, we need to make restrictions that forbid
anyone to deny you these rights or to ask you to surrender the rights.
These restrictions translate to certain responsibilities for you if you
distribute copies of the software, or if you modify it.

  For example, if you distribute copies of such a program, whether
gratis or for a fee, you must give the recipients all the rights that
you have.  You must make sure that they, too, receive or can get the
source code.  And you must show them these terms so they know their
rights.
```

```
  We protect your rights with two steps: (1) copyright the software, and
(2) offer you this license which gives you legal permission to copy,
distribute and/or modify the software.

  Also, for each author's protection and ours, we want to make certain
that everyone understands that there is no warranty for this free
software.  If the software is modified by someone else and passed on, we
want its recipients to know that what they have is not the original, so
that any problems introduced by others will not reflect on the original
authors' reputations.

  Finally, any free program is threatened constantly by software
patents.  We wish to avoid the danger that redistributors of a free
program will individually obtain patent licenses, in effect making the
program proprietary.  To prevent this, we have made it clear that any
patent must be licensed for everyone's free use or not licensed at all.

  The precise terms and conditions for copying, distribution and
modification follow.

                    GNU GENERAL PUBLIC LICENSE
   TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

  0. This License applies to any program or other work which contains
a notice placed by the copyright holder saying it may be distributed
under the terms of this General Public License.  The "Program", below,
refers to any such program or work, and a "work based on the Program"
means either the Program or any derivative work under copyright law:
that is to say, a work containing the Program or a portion of it,
either verbatim or with modifications and/or translated into another
language.  (Hereinafter, translation is included without limitation in
the term "modification".)  Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not
covered by this License; they are outside its scope.  The act of
running the Program is not restricted, and the output from the Program
is covered only if its contents constitute a work based on the
Program (independent of having been made by running the Program).
Whether that is true depends on what the Program does.

  1. You may copy and distribute verbatim copies of the Program's
source code as you receive it, in any medium, provided that you
conspicuously and appropriately publish on each copy an appropriate
copyright notice and disclaimer of warranty; keep intact all the
notices that refer to this License and to the absence of any warranty;
and give any other recipients of the Program a copy of this License
along with the Program.

You may charge a fee for the physical act of transferring a copy, and
you may at your option offer warranty protection in exchange for a fee.

  2. You may modify your copy or copies of the Program or any portion
of it, thus forming a work based on the Program, and copy and
distribute such modifications or work under the terms of Section 1
above, provided that you also meet all of these conditions:

    a) You must cause the modified files to carry prominent notices
```

```
    stating that you changed the files and the date of any change.

    b) You must cause any work that you distribute or publish, that in
    whole or in part contains or is derived from the Program or any
    part thereof, to be licensed as a whole at no charge to all third
    parties under the terms of this License.

    c) If the modified program normally reads commands interactively
    when run, you must cause it, when started running for such
    interactive use in the most ordinary way, to print or display an
    announcement including an appropriate copyright notice and a
    notice that there is no warranty (or else, saying that you provide
    a warranty) and that users may redistribute the program under
    these conditions, and telling the user how to view a copy of this
    License.  (Exception: if the Program itself is interactive but
    does not normally print such an announcement, your work based on
    the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole.  If
identifiable sections of that work are not derived from the Program,
and can be reasonably considered independent and separate works in
themselves, then this License, and its terms, do not apply to those
sections when you distribute them as separate works.  But when you
distribute the same sections as part of a whole which is a work based
on the Program, the distribution of the whole must be on the terms of
this License, whose permissions for other licensees extend to the
entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest
your rights to work written entirely by you; rather, the intent is to
exercise the right to control the distribution of derivative or
collective works based on the Program.

In addition, mere aggregation of another work not based on the Program
with the Program (or with a work based on the Program) on a volume of
a storage or distribution medium does not bring the other work under
the scope of this License.

  3. You may copy and distribute the Program (or a work based on it,
under Section 2) in object code or executable form under the terms of
Sections 1 and 2 above provided that you also do one of the following:

    a) Accompany it with the complete corresponding machine-readable
    source code, which must be distributed under the terms of Sections
    1 and 2 above on a medium customarily used for software interchange; or,

    b) Accompany it with a written offer, valid for at least three
    years, to give any third party, for a charge no more than your
    cost of physically performing source distribution, a complete
    machine-readable copy of the corresponding source code, to be
    distributed under the terms of Sections 1 and 2 above on a medium
    customarily used for software interchange; or,

    c) Accompany it with the information you received as to the offer
    to distribute corresponding source code.  (This alternative is
    allowed only for noncommercial distribution and only if you
    received the program in object code or executable form with such
```

an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under
any particular circumstance, the balance of the section is intended to
apply and the section as a whole is intended to apply in other
circumstances.

It is not the purpose of this section to induce you to infringe any
patents or other property right claims or to contest validity of any
such claims; this section has the sole purpose of protecting the
integrity of the free software distribution system, which is
implemented by public license practices. Many people have made
generous contributions to the wide range of software distributed
through that system in reliance on consistent application of that
system; it is up to the author/donor to decide if he or she is willing
to distribute software through any other system and a licensee cannot
impose that choice.

This section is intended to make thoroughly clear what is believed to
be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in
certain countries either by patents or by copyrighted interfaces, the
original copyright holder who places the Program under this License
may add an explicit geographical distribution limitation excluding
those countries, so that distribution is permitted only in or among
countries not thus excluded. In such case, this License incorporates
the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions
of the General Public License from time to time. Such new versions will
be similar in spirit to the present version, but may differ in detail to
address new problems or concerns.

Each version is given a distinguishing version number. If the Program
specifies a version number of this License which applies to it and "any
later version", you have the option of following the terms and conditions
either of that version or of any later version published by the Free
Software Foundation. If the Program does not specify a version number of
this License, you may choose any version ever published by the Free Software
Foundation.

10. If you wish to incorporate parts of the Program into other free
programs whose distribution conditions are different, write to the author
to ask for permission. For software which is copyrighted by the Free
Software Foundation, write to the Free Software Foundation; we sometimes
make exceptions for this. Our decision will be guided by the two goals
of preserving the free status of all derivatives of our free software and
of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY
FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN
OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES
PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED
OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS
TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE

```
PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING,
REPAIR OR CORRECTION.

  12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING
WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR
REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES,
INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING
OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED
TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY
YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER
PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE
POSSIBILITY OF SUCH DAMAGES.

                     END OF TERMS AND CONDITIONS

        Appendix: How to Apply These Terms to Your New Programs

  If you develop a new program, and you want it to be of the greatest
possible use to the public, the best way to achieve this is to make it
free software which everyone can redistribute and change under these terms.

  To do so, attach the following notices to the program.  It is safest
to attach them to the start of each source file to most effectively
convey the exclusion of warranty; and each file should have at least
the "copyright" line and a pointer to where the full notice is found.


    Copyright (C) 19yy

    This program is free software; you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation; either version 2 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with this program; if not, write to the Free Software
    Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this
when it starts in an interactive mode:

    Gnomovision version 69, Copyright (C) 19yy name of author
    Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type ‘show w’.
    This is free software, and you are welcome to redistribute it
    under certain conditions; type ‘show c’ for details.

The hypothetical commands ‘show w’ and ‘show c’ should show the appropriate
parts of the General Public License.  Of course, the commands you use may
be called something other than ‘show w’ and ‘show c’; they could even be
mouse-clicks or menu items--whatever suits your program.
```

```
You should also get your employer (if you work as a programmer) or your
school, if any, to sign a "copyright disclaimer" for the program, if
necessary.  Here is a sample; alter the names:

  Yoyodyne, Inc., hereby disclaims all copyright interest in the program
  ‘Gnomovision’ (which makes passes at compilers) written by James Hacker.

  , 1 April 1989
  Ty Coon, President of Vice

This General Public License does not permit incorporating your program into
proprietary programs.  If your program is a subroutine library, you may
consider it more useful to permit linking proprietary applications with the
library.  If this is what you want to do, use the GNU Library General
Public License instead of this License.
```

Index

deep copy, 82

explicit sharing, 82
explicitly shared, 82

implicit sharing, 82
implicitly shared, 82

keyboard focus, 20

meta object, 30
moc, 34

Q_OBJECT, 30

reference counting, 82

shallow copy, 82
shared explicitly, 82
shared implicitly, 82
stretch factor, 17