Qt

Cross-platform C++ GUI Application Framework

Technical Overview

v. 1.3

1. Contents

2. Introduction

For many years software producers have been faced with the problem of how to target a market with such diverse operating and window systems. And there is no reason to think this situation is about to change. It is now clear that the early nineties' rumors of the impending death of Unix were exaggerated. The current boom of Linux and its positioning as a competitor on the desktop ensure that the software market will continue to have many different platforms in the foreseeable future. A major challenge in targeting multiple platforms has been the cost of developing and maintaining an application for several different platforms. Because of the inherent differences among these platforms, the porting of an application to a new platform can involve costly redesign and reimplementation.

Qt is a software development application framework that solves the most critical challenges of cross-platform application development and maintenance. The following presentation explains the principles software developers can use with Qt to create single-code applications for end-users on different platforms.

Qt is produced by Trolltech. Qt has been marketed since 1995 and is now being used by such leading companies as HP, IBM, Intel, Siemens, Ericsson and Xerox. Qt widely used on Linux; in fact, it is the basis of the popular KDE desktop environment included in virtually every Linux distribution.

Details of all the functionality Qt provides are outside the scope of this document. For further technical information we refer you to:

- *The Qt Reference Documentation*. Latest version is available on-line at doc.trolltech.com.
- *Programming with Qt*, by Matthias Kalle Dalheimer. O'Reilly, 1999. This book is available in several languages: English, Norwegian, French, Japanese, Korean, Portuguese and German.
- *Teach Yourself Qt Programming in 24 Hours*, by Daniel Solin, SAMS, 2000
- *KDE- und Qt-Programmierung*, by Burkhard Lehner, Addison-Wesley Longman Verlag, 2000. Currently available only in German.

An updated list of all available documentation is available on-line at www.trolltech.com/developer/literature.

Qt is a continuously evolving toolkit. The following summarizes the main features of the current version of Qt, which is version 2.2. Further information about Qt is available at the Trolltech web site:

www.trolltech.com

3. Architecture

Qt is a cross-platform C++ application framework. It is implemented as a class library and provides a rich API (Application Programmer's Interface) for application developers. Qt offers a wide spectrum of useful functionality but focuses mainly on the GUI (Graphical User Interface). Thus, for application developers *Qt replaces Motif, MFC, and/or other GUI toolkits.*

3.1. Cross-Platform Development

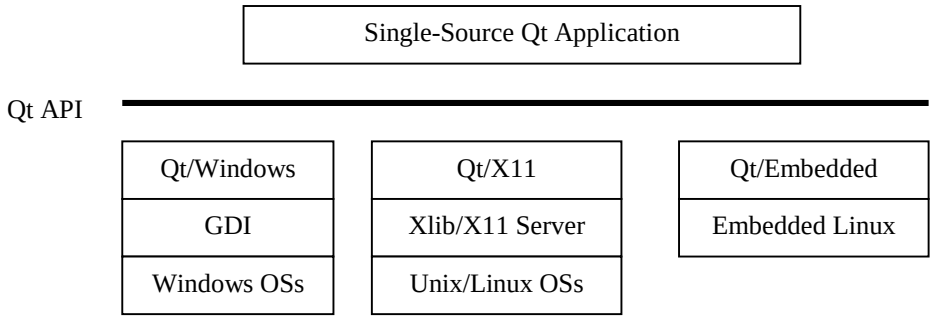
Qt is cross-platform: the Qt class library is implemented for several different operating and window systems., and the API is identical for all platforms. This means that an application written with Qt on one platform can be made to run on another simply by recompiling it on the new platform and linking it with the Qt library for that platform. In fact, the Qt library is binary compatible on all supported Windows variants. Thus, *with Qt, software producers can develop and maintain an application for multiple platforms by developing and maintaining a single application source code base.*

Qt is currently implemented for three main groups of operating systems:

- Unix - including Linux, HP-UX, Sun Solaris, Digital Unix, SGI Irix, IBM AIX, SCO Unix, and several BSD variants. The Qt library is implemented using the X11 libraries and the X Window system.
- Windows - including Windows 95, 98, NT and 2000. The Qt library is implemented using the Windows GDI API and the Microsoft Windows window system.
- Qt/Embedded includes a complete window system and can be easily targeted to any display and input hardware.

Qt platforms can be depicted as shown in Figure 1 below:

Figure 1: Qt Platforms



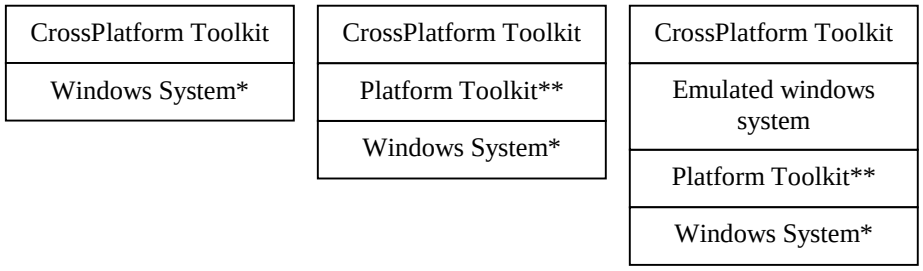
Implementations for additional operating and window systems are being developed.

The Qt library code is extremely portable. All major hardware architectures for the various operating systems are supported, including 64-bit systems. The Unix / X11 implementation is in commercial use most successfully not only on the above-mentioned operating systems, but on real-time operating systems like QNX and VxWorks, and on OS/2 Warp using the XFree86 X server. Qt/Embedded will be available for Embedded Linux and FreeBSD and NetBSD; additional ports are being planned.

3.2. API Layering

When designing a cross-platform library like Qt, one has a choice of three conceptually different architectures: Emulating, Layered, or Native-API Emulating Architecture. These models are depicted in Figure 2 below:

Figure 2: Emulating versus Layered Architecture



* The Windows System represents the lowest graphics level provided by the respective platform; e.g. GDI on Windows or Xlib on Unix.

** The Platform Toolkit represents the native GUI components (widgets) offered by the platform; e.g. MFC on Windows, and Motif on Unix.

In the cross-platform GUI library industry, the pros and cons of the Emulating versus Layered Architectures have been argued for years. Initially, layered architecture was preferred, and when this industry boomed for the first time in the late eighties most products were layered. However, during the nineties many of these products failed, and now they are no longer maintained or supported.

Qt implements the single-layered architecture known as the Emulating Architecture. The cross-platform library must then implement all the necessary widgets using its own API and emulate the look and feel of the underlying platform.

Emulation Architecture should not be confused with the Native-API Emulation Architecture libraries that emulate one native API on tope of another, e.g an MFC emulator for Unix/X11. (Native-API Emulation architecture is very different from the other two; see footnote below.[1])

The main issues in the layered versus emulating architecture discussion will be presented in the following sections, together with the reasons why the emulating solution was chosen for Qt.

[1] Native-API Emulation Libraries

As referred to earlier, another option for a cross-platform library is to emulate the native API of one platform on another, such as the architecture of the Unix/X11 MFC emulation libraries. This design adapts a legacy application (built using the native API) to quickly run on another platform by avoiding the need to reimplement it using the special API of the cross-platform toolkit.

The negative features of this design are as follow:

- It is really a special kind of layered design, with all the disadvantages of that architecture discussed above. , Because yet another layer is introduced, performance can be most unreliable.
- Application programmers often find the existing native APIs, such as Motif and MFC, to be complex and cumbersome to use. (Qt's design to offers application programmers an intuitive and truly object-oriented API.)
- Real-world native applications are seldom well-behaved in the sense that they depend on undocumented quirks of the native API, and they bypass the native API to achieve special effects by accessing the lower layers directly, etc. Attempting to emulate the native API closely enough to handle such code is extremely difficult.

3.3. Look and Feel

The main argument in favor of the layered approach is that it is the only way to achieve exact conformance with the native look and feel. As the name implies, an emulating toolkit must emulate the native GUI elements, and this emulation will unavoidably be imperfect. The strength of this argument rests on two assumptions:

- Users will resent applications with even the slightest variations in look and feel.
- The emulating toolkit's task of keeping up with the changes and developments in the OS's native look and feel is insurmountable.

Although both assumptions were relevant five to ten years ago, the situation differs now. Previously there were many contenders for the position of standard GUI: Windows, OS/2 Presentation Manager, Macintosh, Motif, among others. The ensuing "religious wars" made users extremely sensitive to look and feel issues. Today, this battle is no longer relevant; the introduction of Java and other technologies diffuses the strict look and feel standards. For example, Microsoft itself introduces minor changes in their products' look and feel with every major version of Windows and Microsoft Office. Since applications do not keep up with this development (not even Microsoft's own; compare, for example, the look and feel of the menus and file dialogs of "Notepad" and "Word"), users have become accustomed to slight deviations in look and feel between applications. Though not desirable, such deviations are no longer perceived to be anything like a show-stopper for an application.

The second argument is also becoming increasingly irrelevant. The late eighties and early nineties was a time of rapid development of look and feel, and major changes like the change from Windows 3.1 to Windows 95 took place. Keeping an emulation toolkit up-to-date in this period was a difficult task indeed. Since then, however, the pace of native look and feel development has slowed considerably. For example, as mentioned above, no Windows version since Windows 95 has introduced anything more than minor extensions and variations. This trend will likely continue because of the sheer number of users who by now have been trained in, and become accustomed to, the industry-standard look-and-feels. Thus, it is now feasible to keep an emulating toolkit quite up-to-date.

When considering the look and feel argument it should also be noted that even a layered toolkit must use emulation if it wants to provide GUI elements not offered by the native GUI. The alternative is to be a so-called "least common denominator" toolkit, which is not a satisfactory solution. Customized GUI elements must also use emulation, even in a layered toolkit.

Qt solves the look and feel issue by doing close emulation of the native look and feel standard. All visual elements in Qt are implemented with a dynamic look and feel. This means that they will present themselves to the user somewhat differently, depending on the application's currently selected look and feel style, or "theme". *A Qt-based application can employ any look and feel style on any platform*, and the style can even be changed at run-time. Qt provides the following default styles:

- Motif style – emulates the classic Motif look and feel. This is the default style of Qt-based applications running under X11.
- CDE style – a variation of the Motif style which emulates the lighter Motif look and feel that has become popular in the recent years.
- Windows style – closely emulates the Windows look and feel. This is the default style of Qt-based applications running under Windows.
- SGI – closely emulates another popular and lighter Motif look and feel
- Motif Plus – emulates the GTK look and feel
- Platinum – emulates the GUI found on the MacOS

An API is also provided to implement customized styles. This means that for applications with special demands as to visual appearance, e.g. a kiosk application, the programmer can easily implement a customized look and feel. All visual elements in Qt will then present themselves using this customized look and feel style. In contrast, a layered architecture cannot provide the option of using a non-native style, and it cannot let the programmer define customized styles.

3.4. Performance

It is difficult to make a general conclusion about the performance of the two types of cross-platform toolkit architecture. Proponents of the emulating approach will argue that an application built with a layered toolkit is excessively bulky since it must include not only the toolkit itself but also all the layers below. Proponents of the layered approach, on the other hand, will contend that applications built with emulating toolkits become bulky since they must include a replacement for the native GUI functionality already installed on the target system.

As for execution speed, the emulating approach has some advantages. Firstly, GUI function calls pass through fewer layers. Secondly, it avoids the typical bulkiness and sluggishness of the native GUI libraries (e.g. Motif and MFC).

Qt is a relatively lightweight library. This is particularly important in situations where memory resources are scarce. For example, for an embedded Unix system, a Qt-based application will typically be much less resource-demanding than an equivalent application built with a layered toolkit + Motif + Xt (or even just Motif + Xt).

The latest Qt version includes a new configuration mechanism with which the application developer can customize the size of the Qt library by leaving out features. If the current application does not need some feature normally offered by Qt, simply use this new mechanism to omit it during compilation and thus control the size of the resulting library. For lightweight applications it is possible to reduce the footprint of Qt to around 700K. This, in addition to the fact that Qt/Embedded does not need X11, can reduce the cost of offering rich, high quality graphical user interfaces on embedded devices.

3.5. Maintenance

The implementation of a layered toolkit is tightly bound to the native GUI API of each of the supported platforms. This creates a maintenance load on the toolkit developers whenever new versions of the native GUI APIs are released. An emulating toolkit like Qt, on the other hand, uses only a small set of platform functions: basic graphics and user input routines. Furthermore, these functions are on a lower level which is less likely to undergo version changes. Qt is also designed so that large parts of its implementation (including the widget set and tool classes) are platform independent, relying only on the platform-dependent Qt kernel.

3.6. API design

Some important design features of Qt are:

- Effective use of object-oriented principles. For example, all widget classes (both ready-made and customized) inherit from the same basic QWidget class. Thus, all widgets have a large set of common, immediately usable functions.
- Qt is not a “least common denominator” toolkit. Qt provides features not found in all supported window systems by implementing them internally. For example, Qt lets applications draw rotated and transformed text. Of the platforms currently supported by Qt, only Windows NT provides this functionality natively, but Qt implements it internally for Windows 95/98/2000 and X11.
- Run-time flexibility. Qt-based applications do not depend on any external, static resource files or similar. All aspects of the GUI can be changed or added at run-time.

These features optimize run-time performance and memory usage while minimizing the footprint.

3.7. Efficiency

Run-time efficiency and performance is a central design objective of Qt implementation. For example, Qt's graphics drawing functionality is hand-optimized for speed, using internally implemented algorithms instead of the native drawing engine functions where experiments have determined that the latter is slower.

One of the techniques Qt uses to improve application performance is reference-counted, copy-on-write sharing. This means that many classes are implemented so that copies of the same object will share the same data in memory. This saves unnecessary copying of the data and reduces memory demands of the application as a whole. This feature is especially effective when applied to classes containing large data amounts, such as pixmaps and images, and frequently used classes such as strings. All such classes in Qt are shared.

3.8. Component Programming

In object-oriented software development, it is desirable to structure the application code in independent, reusable components. This principle is known as component programming. Qt helps application programmers in this task with a special inter-object communication mechanism called *signals and slots*. This mechanism allows objects to emit anonymous signals that cause slot functions in other objects to be executed. It is a form of inter-object communication mechanism not unlike Motif callbacks and MFC message maps, but with some important advantages that will be detailed below.

The signal-slot mechanism consists of the following constructs:

- All classes defining either signals or slots must inherit from the Qt base class QObject.
- A QObject class may define any of its (otherwise normal) member functions to be slots.
- A QObject class may define that it is able to emit certain signals. Signals have a name and a parameter list, like member functions.
- A signal of one QObject may be connected to a slot of another QObject. If the signals and slots are declared public, this connection can even be done by a third object.
- A QObject may at any time choose to emit a signal.

The effect of these constructs is that every time a QObject emits a signal, the slot function of the QObject(s) it has been connected to is executed immediately. Parameter values are passed from the emitting object to the slot functions. Emitting a signal is thus like a function call, but with the critical difference that the emitting (calling) QObject does not need to know which slot functions (if any) of which QObjects (if any) will be executed. This makes it possible to design application-independent, reusable classes.

Note that all QWidgets are also QObjects with predefined signals and slots ready to be used. The logic of the application is controlled by the way the application developer decides to connect signals and slots together. By subclassing virtual functions found in Qt, the application developer may also customize the look and behavior of existing widgets.

A signal may be connected to any number of slot functions, and a slot function may have any number of signals connected to it. Connections can be established and removed at run-time. Any number and types of parameters may be passed with the signal, just as with a normal function call. The signal-slot mechanism provides full-parameter type safety. If an application tries to connect a signal to a slot with mismatching parameter types, a warning message is issued and the connection is ignored. Superfluous signal parameters are silently ignored; for example, a signal with an integer parameter followed by a string parameter may be connected to slot functions that take no parameters, or only an integer parameter, or an integer parameter followed by a string parameter.

Qt's signal-slot mechanism replaces the traditional callback mechanisms of older toolkits. An important advantage of the signal-slot mechanism is that it is type-safe; mismatches between the

parameter types of the signal and the slot are easily resolved. Such mismatches in callback functions in other toolkits invariably lead to run-time failures (segmentation faults) and hard application termination.

Here is an example of the typical use of the signal-slot mechanism. Assume an application design calls for a dialog box that gets closed when the user clicks its "OK" button. The Qt programmer implements this by the classes QDialog and QPushButton. The QPushButton class has a signal called `clicked()` that gets emitted when the user operates the button. The QDialog class has a slot function called `accept()` that closes the dialog. Thus, the programmer can achieve the desired functionality by simply connecting the `clicked()` signal of the QPushButton object to the `accept()` slot of the QDialog object. The code looks like this:

```
// create the objects
QDialog *d = new QDialog(...);
QPushButton *b = new QPushButton(...);

// connect the signal to the slot
connect (b, SIGNAL(clicked()), d, SLOT(accept()));
```

4. Internationalization

Qt allows applications to use any language and character set. It is easy to switch language, even at run-time. With this feature, the application developer offers the user a choice of languages to use. In this way Qt empowers both the application developer and the application's end users.

4.1. Unicode

With Qt, applications can use international (i.e. non-ASCII) character sets. For text operations in particular, Qt provides the QString class, which contains a text string in the 16-bit Unicode standard encoding; Qt is 16-bit clean throughout. The Qt kernel uses the QString class for all internal text operations, as well as for all API functions that take or return text parameters. This includes all text labels of widgets such as labels on push buttons, menu items, content of line edits, etc.

A note about performance: QString is highly optimized; in our tests of moving real-world applications from 8-bit to 16-bit strings, no significant performance penalty was observed.

Qt supports keyboard input and screen output of Unicode text provided by the underlying window system. Screen output requires the appropriate font(s) to be installed. These fonts need not be Unicode-encoded; Qt provides codecs between Unicode and many of the common font encodings. Customized codecs can also be added.

All application text I/O, e.g. to/from files, may be passed through a text codec, which translates between the preferred local format and the Unicode standard format used internally. Codecs for a number of commonly used locales are provided, as well as an API for implementation of custom codecs.

4.2. Localization

Qt provides support for creating localized applications, i.e. applications that can choose as late as run-time in which language to display all the user-visible texts. The choice may be made automatically based on the user's locale setting, or explicitly by the application. This is achieved by offering the user a language selection dialog on startup.

Building a Qt application prepared for localization is easy: the programmer simply passes all user-visible texts through Qt's `tr()` ("translate") function before passing them to Qt for display. For example, the non-localized application code to make a push button display the text label "proceed" would be:

```
myPushButton->setText( "Proceed" );
```

While the localization-prepared version would be:

```
myPushButton->setText( tr( "Proceed" ) );
```

The translate function will do a lookup in the current translation table and return the text string (translation) corresponding to the argument.

An extra feature of the translate function is that if no translation table is installed, it will simply return its argument. This means that a localization-prepared application will run equally well with or without translation tables. This is important during application development when translation tables often have not yet been produced, or when releasing the first version(s) of an application which is planned to be localized in later versions.

Qt also provides tools to assist application developers building and maintain translation tables. One tool, `findtr()`, searches the application source code for strings that need translation and

produces a formatted text file with empty areas where the application translators will simply fill in the required translations. Another tool, `msg2qm()`, converts these text files to the binary, hashed translation table files that are used by Qt for lookup at run-time. A third tool, `mergetr()`, helps to merge existing translation files when the application has been extended or modified so that new strings that need translation have been added.

The following example shows how easily Qt adapts your application to customers using any language. After having written the source using the `tr()` function for all visible texts, use the `findTr` tool to generate the original translation files. The translation file template for mywidget.cpp would be mywidget.po, and part of it could then look like this:

```
#: mywidget.cpp:28
msgid “MyWidget:.E&xit”
msgstr “”

#: mywidget.cpp:41
msgid “MyWidget::Perspective”
msgstr “”
```

Copy mywidget.po to mywidget_jp.po and insert – for example, Japanese – signs as appropriate. Coding then looks something like this:

```
#: mywidget.cpp:28
msgid “MyWidget::E&xit”
msgstr “エグジット”

#: mywidget.cpp:41
msgid “MyWidget::Perspective”
msgstr “      ”
```

The only step needed now is to use the msg2qm() tool to generate the binary translation file, mywidget_jp.qm, used run-time by Qt. The same sequence can naturally be repeated for any language.

With the appropriate translation files and font sets installed, Qt allows the application’s language to be changed at run-time. Simply create a QTranslator object, load the appropriate translation file, and apply the translator to the current QApplication. In code it looks like this:

```
QTranslator *translator = new QTranslator(0);
QString lang(“mywidget_jp.qm”);
translator->load(lang, “.”);
qApp->installTranslator(translator);
```

All visible text will now be displayed using the Japanese translations – to the delight of your Japanese users! *Localization is another example of powerful but often difficult-to-implement concepts made simple by Qt.*

5. Graphical User Interfaces (GUI)

Few, if any, aspects of computing have contributed more to the popular acceptance of computers than the invention of the Graphical User Interface. Not even the most optimistic scientists at Xerox could have foreseen how their research would influence so many aspects of life in only 30 years! But what components make up a modern graphical user interface and how does Qt implement such elements?

5.1. Basic Concepts

Qt's graphical user interface elements are called widgets. Push buttons, scroll bars, and menus are examples of widgets. To the programmer, a widget is an object (instance) of a C++ widget class. For example, you create a push button by instantiating an object of the QPushButton class. All widget classes inherit (directly or indirectly) from the fundamental widget class QWidget.

Many GUI toolkits operate with two different types of GUI elements:

- controls – basic elements like buttons and scroll bars
- containers – elements that contain controls, such as dialogs and application windows.

Qt widgets are more flexible because in this application there is no fundamental difference between containers and controls; any widget may function as one or the other. Containment is expressed in a parent-child relationship; a Qt widget that contains other widgets is called the parent of the contained widgets.

Each Qt widget class provides an API to access the contents and behavior of the widget. For example, the QPopupMenu widget class provides an `insertItem()` method that adds a new item to the menu. Qt's signal-slot mechanism is typically used for the interface to the run-time behavior of the widget. QPopupMenu will emit a certain signal whenever a menu item has been selected. Note that the QPopupMenu signal is emitted independently of how the menu item got selected, i.e. whether by mouse click, keyboard accelerator, or programmatically from another part of the application. This makes it easier to ensure internal consistency in the application.

5.2. User Interface Composition

With Qt, making a normal application window is straightforward. The application programmer starts by creating an object of a suitable container widget class. The various controls are then added to this widget by creating widget objects as children of the container widget. The precise graphical layout of the child widgets will typically be arranged by a layout manager (see below). Lastly, the application programmer implements the functionality of the window by connecting the child widgets' signals and slots to each other, and to the application code.

Qt provides a large set of ready-to-use widget classes from which user interfaces can be built. These classes include all the common GUI controls typically found in modern user interfaces, such as buttons, scroll bars, tool bars, explorer-style hierarchical list views, etc. Normal user interfaces can thus be constructed rapidly by composing standard widgets as described above. (A complete listing of the standard widget classes is given in **Appendix 1**.)

5.3. Layout Management

When implementing the visual appearance of a GUI, one of the main tasks is to decide the positions and sizes of the child widgets within their parent's area. Although it is possible to hard-code static coordinate values for all widgets, this approach is usually not satisfactory for anything but the simplest applications for the following reasons:

- Most applications will want to allow the user to resize the application window while retaining window contents. This might produce unwanted results if the coordinates are static.
- For localized applications or other applications where the contents of otherwise static widgets can change dynamically at run-time, suitable coordinate values cannot be known in advance.
- Similarly, applications that want to honor the user's preferred font setting cannot know in advance how much space is required to display its widgets using that font.
- It is a time-consuming and tedious task for the programmer to tune the widgets' positions and sizes so that they align and give the desired aesthetic effect. Maintenance is also a problem, since the whole layout must be manually reimplemented whenever widgets are added or removed.

Qt overcomes the above problems by providing a mechanism for automatic widget layout management. Using an API, a widget creates a layout manager object to assign positions and sizes for the child widgets. The layout manager does this by dividing the widgets' available area into virtual cells (as many as there are child widgets), and placing one child widget in each cell. When the widget is resized or a child widget's size requirements change, the layout manager automatically recalculates the layout and moves and resizes all the child widgets to fit.

Qt provides two basic layout manager classes:

- QBoxLayout divides the available space into a stack of cells (horizontal or vertical).
- QGridLayout divides the available space into an n x m grid of cells.

Customized layout manager classes may also be added. Note that instead of a child widget, a cell may contain another layout manager object which. in turn. manages other child widgets. Automatic layout of very complex user interfaces can thus be readily achieved by building a nested structure of layout managers.

Each widget class specifies its own layout requirements:

- A widget may specify a preferred size for itself.
- A widget may specify a minimum size it needs to display itself in a satisfactory manner. For example, a push button will in this way ask not to be made so small that it cannot paint its label and the surrounding button frame.
- A widget may specify that it should not be stretched out more than its preferred size, or that it should be stretched in only one direction. For example, a vertical scroll bar will specify that it can be stretched vertically but not horizontally, since the latter would ruin its visual appearance.

Naturally, all of Qt's standard widgets provide sensible, run-time default values calculated to use the widgets' current contents and state for all the above-mentioned constraints. If the contents of a widget change while the program is running, the layout will be recalculated automatically to fit the new size of the widget.

The layout algorithm is tuned as follows:

- A stretch factor is assigned to each cell to determine what ratio of the available, superfluous space the layout manager will assign to it.
- The widths of the blank borders around and between the cells are changed. Extra blank space (stretching or non-stretching) may be added.
- The alignment (left/right/ top/bottom/center) of the child widget within the cell is specified.
- The maximum and/or minimum size of the child widget is set explicitly.

Virtually any layout behavior can be obtained by applying the above adjustments to the standard layout algorithm.

5.4. Customizing Widgets

A central design feature of Qt's widget system is extensibility. This is important; experience shows that a fixed set of static widget classes cannot cover all requirements of a real-world application. GUI toolkit designers try to foresee various demands that applications may have and to provide the necessary functionality in the widget classes - indeed, Qt's standard widget classes are similarly designed. But this is no substitute for enabling application developers to easily customize widget classes or to design their own widget classes from scratch.

Qt makes it very easy for application programmers to create customized widget classes. The application programmer simply makes a new C++ class that inherits QWidget (directly or indirectly) - there are no resource files to be edited or mandatory methods to implement (except for the constructor, as the C++ syntax demands). Depending on what the widget shall do, the programmer can choose to implement any of QWidget's virtual methods in order to receive events, customize look and feel, etc.

Custom widgets have many effective uses:

Application Data Presentation
Custom widget classes can be used to implement fundamental and unique graphical interface applications, such as a process control application's graphs of data samples, a word processor's WYSIWYG window, or a network management application's visual presentation of the network topology. The customized widget class uses Qt's graphics API for the presentation and the event system to implement user navigation and data manipulation.

Qt offers some special widget classes as the basis for implementing customized widgets:

- QScrollView provides subclasses with a framework for building widget classes that display just part of a potentially much larger virtual canvas area. If necessary, scroll bars are automatically provided. Contents may consist of graphics directly drawn and/or child widgets. For example, a network management application may use direct graphics to draw a representation of the network topology, and let the network nodes be represented by push button widgets that the user can click to get a pop-up window with the current status of that node.
- QTable provides a framework for building widgets that display data in tabular (spread sheet style) format.
- QCanvas - see section 7.5

Tuning the Behavior of Standard Controls
An application often needs a GUI control that is almost, but not quite, the same as one of the standard controls. With Qt, tuning behavior is made easy using the power of inheritance and virtual functions in C++.

For example, an application may require a spin box widget that operates on dates instead of integers, or a slider widget that will jump to a predefined value when the user presses a function key. By making a custom widget class that inherits from the standard widget class, the application programmer has almost unlimited power to modify its behavior to fit the application requirements. This is because all key methods in the widget classes are C++ virtual functions, so the custom widget class can re-implement them, overriding or amending to the original implementation.

Custom Controls

Applications sometimes require new kinds of basic user interaction elements. For example, a word processor may require a ruler so the user can specify tab stops in a word processor. The application programmer can achieve precisely the desired behavior by creating a customized widget class. All the functionality of Qt used to implement its standard widget classes is also available to the custom widget class programmers.

Control widget classes may also be implemented with the help of child widgets. For example, Qt's combo box widget class is implemented using a line edit widget and a pop-up menu widget.

6. Visual Development

Although the layout management offered by the Qt API does help the user overcome most of the problems noted earlier (see section 5.3), it is still cumbersome and time-consuming to develop a large number of dialogs and widgets working only with the C++ API. Developers need better tools to speed the development and maintenance of user interfaces.

IDE and RAD Limitations

Integrated Development Environments (IDE) and Rapid Application Development (RAD) tools (such as Borland Delphi and Microsoft Visual Basic) try to fulfill this need. But each new IDE or RAD imposes new burdens on application developer by presenting new, often very complex user interface to become familiar with. Time and effort must be invested to understand the logic and operation of the new application. For example, how do you set up the environment and preferences properly?

The greatest negative is that most of these applications try to be too clever and lock their users into their own technology and products. This makes sense from a strictly business point of view, but it is often not the optimal solution for the developer. For instance, in Microsoft Studio programmers are expected to use the accompanying text editor as well as the compiler. Traditionally, IDE and RAD tools tend to offer poor support for integrating with the developer's favorite text editor or compiler.

Qt Designer

Tolltech's Qt 2.2 introduces Qt Designer: an application designed specifically to make construction and maintenance of graphical user interfaces faster and easier. With respect to the problems described above, let us clarify what Qt Designer can and cannot do.

Qt Designer:

- offers the user a point and click, Drag and Drop, WYSIWYG interface to simplify GUI creation and layout
- allows easy access to widget properties
- provides easy access to all the standard widgets as well as to Qt's layout management
- employs a vendor-neutral, fully documented XML-format for persistent storage
- facilitates editing and regeneration of forms by employing a two-layered C++ class hierarchy. Since the application developer works with subclasses, regenerating the forms will not interfere with work done previously.
- allows different groups to work simultaneously on design and implementation (often a requirement in organizations with dedicated design departments).

Qt Designer does not:

- generate C++ code. A separate tool, the User Interface Compiler, generates C++ files from the User Interface files Qt Designer uses for persistent storage. This tool could easily be replaced if the user prefers to work with another XML-to-C++ converting tool.
- force the developer to use a particular text editor, compiler or debugger. It integrates easily with whatever tools the developer prefers to working.
- work off source code. In fact, Qt Designer is not capable of loading C++ source. (For reverse engineering, it would of course be possible to make a tool capable of creating the required XML files from C++ source files.)

From the above it should be clear that Qt Designer is not just another IDE. Rather, *Qt Designer is a specialized tool which can integrate with the application developer's own preferred development environment.* This is done in a visual manner which greatly speeds up the specific task of creating and maintaining graphical user interfaces. To the developer, Qt Designer is a simple, but very powerful tool.

Customized Widgets
In its present version Qt Designer provides only rudimentary support for customized (user) widgets. But the next version will offer a plug-in system for seamless support of customized widgets.

As Qt Designer is now, you only see a box to represent customized widgets. You can modify the properties the user has given, use the signal/slots for connections given by the user's definition and supply the `sizeHint()`and `sizePolicy()` functions which Qt uses for layouts While this is sufficient for many cases, the next version will provide a plug-in system so it is not longer necessary to specify this information. The next version will allow dynamical loading of any customized widget, which Qt Designer will then treat as any other widget.

7. Graphics

Qt has a mature 2D graphics API. A special feature is the QPainter class which offers basic graphics functionality. It provides a high-level drawing engine with commands for drawing lines, polygons, ellipses, splines, images, pixmaps, texts, etc. The Qt graphics system includes the following features:

7.1. Device Independent Graphics

Qt supports graphics drawing to screen (i.e. widgets), printers, pixmaps and pictures (known as a meta-file in Windows terminology). Qt hides the intrinsic differences between these devices from the application programmer. In Qt, they are all paint devices.

A QPainter operates on a paint device, and the application using the QPainter need not be concerned about whether this QPainter is currently drawing on a widget or on a printer. The drawing API is totally device-independent. This is practical for many tasks. For instance, applications may use the same drawing routine for screen output as for print output; this is achieved by simply making the drawing routine take the QPainter as a parameter, and then passing one QPainter operating on a widget for screen drawing and one QPainter operating on a printer for printer drawing.

7.2. Special Paint Devices

In addition to widgets and printers, QPainter supports drawing to two special devices:

QPixmap
A pixmap is an off-screen memory frame area, i.e. a two-dimensional array of pixel values. Qt provides fast bitblt() (bit block transfer) operations to move pixels between widgets and pixmaps. For example, if an application is going to display complex static graphics on screen, it makes sense to draw the graphics into a pixmap and then draw the pixmap to the screen later. This technique, known as double-buffering, is more efficient since the complex drawing need only be performed once. It can also be used to eliminate screen flicker.

Pixmaps are also handy for storing graphics to file for later retrieval or for other transfer of image data.

QPicture
A picture is actually a stored sequence of drawing operations. Pictures are useful to store graphics for re-display at a different magnification level, for example. Zooming in on a pixmap will only magnify the individual pixels - but zooming in on a picture will recreate the drawing as if it had been drawn at that scale originally.

7.3. The 2D Graphics API

QPainter is implemented using the drawing operations of the underlying window system, e.g. Xlib on Unix/X11 and Windows GDI on Windows. Features the underlying system lacks (e.g. drawing transformations on Windows 95/98 and X11) are implemented within Qt itself.

Color handling
Qt provides a separate class to specify color for drawing operations. Colors are specified as RGB or HSV values, or as a name from the web standard (e.g. “steelblue”, “green4”, etc.). On systems with limited color ranges (e.g. 8-bit displays) Qt automatically allocates colors in the system palette, so Qt-based programs need not be specially adapted to run on such systems.

The Drawing Style
Qt provides the QPen class to specify the desired graphics attributes of lines, polygon outlines, etc., the application developer uses QPen to control line color, width, and stipple pattern.

Qt provides the QBrush class for the fill style of polygons, ellipses etc., including color and pattern. QBrush provides a set of predefined patterns, but a custom fill pattern (specified as a pixmap) can also be used.

Transformations
QPainter provides full support for transformations, i.e. scaling, rotating, etc. A QPainter's world transformation specifies how the world coordinates (i.e. the parameter values given to the drawRect() method for example) will be transformed into logical coordinates. The view transformation specifies how these logical coordinates, in turn, will be transformed into the physical coordinates of the paint device.

Qt supports a general transformation matrix for the world transform, in which all forms of coordinate translation, scaling, rotation and shearing can be performed. A separate transformation matrix class is also provided, but QPainter has convenient functions to specify the most common transformations directly.

Qt allows setting the origin and extent of the drawing window and the drawing viewport for the view transform. The drawing viewport determines the logical coordinate system; specifying this to, e.g., 1000 x 1000 gives the application programmer a 1000 x 1000 drawing area independent of the size of the underlying physical device. The drawing window, on the other hand, determines the rectangle of the physical device that the logical coordinates will be mapped down into.

All drawing operations provided by QPainter may also be performed with world and/or view transformation applied, including text and pixmap drawing.

Clipping
QPainter allows clipping to a rectangle or a more general region composed of a set of rectangles, polygons, ellipses, and bitmaps. The composition can be made as unions, intersections and/or subtractions.

Text Drawing and Fonts
QPainter provides two text drawing methods: a simple function for drawing a given text at a given x,y coordinate, and a more complex function allowing the specification of :

- a rectangle Qt should fit the text into
- how to align the text within the rectangle (top, bottom, flush left, center, flush right)
- whether Qt should break the text into lines to fit the width of the rectangle

A separate class, QFont, is provided to specify the font. All fonts installed in the underlying window system may be used to draw text in Qt. A font may be selected by specifying any or all of its name, size, weight (bold), slant (italic), and character set. Qt will provide the closest matching available font. Font sizes can be given as logical (dpi) or pixel sizes.

A number of international character sets are supported, including ISO_8859-1 - ISO_8859-15 (Latin1-Latin9), KOI8R, Japanese and various other Asian character sets.

7.4. Image Handling

Qt supports input, output, and manipulation of images in several formats, including PNG, BMP (Windows bitmap), XBM (X11 bitmap), XPM (X11 pixmap), PNM (P1-P6), and optionally GIF (note that including GIF support may require patent licensing from Unisys). All platforms support all image formats, e.g. BMP on both Windows and Unix. Image formats are auto-detected on reading.

The Qt ImageIO Extension library adds support for the JPEG format; application programmers can also add support for custom formats. The image formats added with the ImageIO Extension become fully integrated with Qt's image handling system, just like the internally supported formats.

Once read into an application, an image is stored in a QImage object. This class provides an API that allows manipulating the image data in a hardware-independent manner. This means that applications using QImage for image manipulation can easily be designed to function independently of the screen depth and byte-ordering (endianess) properties of the hardware they run on. QImage also provides direct access to the image data (memory block), for speed-critical operations.

QImage supports images of 32-, 8-, or 1-bit depth. Images with other depths are automatically converted to the next higher supported depth. For 8 or 1 bit deep images, a color palette is provided, which also may be manipulated. Depth conversion methods are provided, including optional dithering when converting to a lower depth.

For each pixel in a 32-bit deep QImage, an 8-bit value is stored for each of the red, green, blue and alpha components. The optional alpha component may be used for custom image operations relating to image transparency, blending, etc.

Qt also supports reading of animation image formats, with asynchronous (e.g. frame-by-frame) reading for interleaved reading and display. The QMovie class provides easy handling of animations.

7.5. Canvas

Qt version 2.2 also introduces the QCanvas class. This contains any number of QCanvasItems and has multiple CanvasView widgets observing any part of the canvas. QCanvasItems represent drawing primitives such as pixmaps, rectangles, lines and texts, and can be moved, hidden, and tested for collision with other items. They have selected, enabled, and active state flags which subclasses may use to adjust appearance or behavior.

A canvas containing many items is different from a widget containing many sub-widget:

- Items are drawn much faster than widgets, especially when non-rectangular.
- Items use less memory than widgets.
- You can do efficient item-to-item hit tests("collision detection") with items in a canvas.
- Finding items in an area is efficient.
- You can have multiple views of a canvas.

Widgets naturally offer richer functionality with respect to hierarchies, events, and layout.

Each canvas has a solid background and a foreground, and all items are drawn in between. Qt provides a rich API, offering control of the appearance of each layer. QCanvas also simplifies animation by enabling the user to control the speed and direction of movement of each QCanvasItem. Finally, QCanvas offers collision testing. Using the API, it can easily be determined whether any items collide with a point, a rectangle or with other items.

7.6. 3D Graphics

Qt does not itself offer 3D graphics functionality, but integration with 3rd party 3D libraries is provided. Many Qt users employ these extensions to offer exciting 3D solutions for their clients and customers.

It should be noted that these integration packages do not depend on special support within Qt itself; the ordinary Qt API provides the necessary general low-level access functions. Application programmers can thus build custom integration packages to other libraries.

OpenGL
The Qt OpenGL Extension library provides integration with OpenGL. It allows application developers to build data display widgets where the contents are drawn using the native OpenGL library instead of Qt's 2D graphics code. The Qt OpenGL Extension also provides a platform-independent C++ wrapper API around the platform-specific C APIs GLX and WGL.

HOOPS
HOOPS is a high-level, cross-platform, object-oriented graphics subsystem that simplifies the design, development and maintenance of high-performance, interactive 2D and 3D graphics applications. It is a product of Tech Soft America, who offer a HOOPS-Qt integration package.

8. Tool Classes

Qt is more than a GUI toolkit: it is an application framework. In addition to the GUI functionality, Qt provides application developers with a comprehensive set of generally useful tool classes.

Some of Qt's tool classes provide functionality similar to the C++ standard library and the STL. In particular, Qt's tool classes are preferred for most Qt-based applications because of their special features:

- Portability: Qt classes are portable to a wide range of platforms and compilers. Many of these platforms lack a functional and standard-conformant STL implementation. By using Qt classes, application programmers avoid such portability issues.
- Cross-platform data exchange: Qt's classes for data I/O provide platform- and architecture-independence, so that even binary data can be successfully exchanged between one platform and another (this is not the case with standard I/O).
- Internationalization: Qt's classes for text handling and I/O are Unicode-based and thus fully prepared for internationalization (again, this is not true for the standard classes).

Application developers may, however, choose to use the standard library and/or the STL instead of, or in combination with, the Qt tool classes; tool classes may coexist in the same application without problems, and data conversion between tool sets is straightforward.

8.1. Operating System Services

The task of making an application truly portable involves more than giving it a cross-platform GUI. Real-world applications will always need to access various operating system services which typically have different, incompatible APIs on different operating systems.

Qt provides OS-independent encapsulations of the most commonly used OS services. Thus, by using the Qt API instead of the native OS API, Qt-based applications can be immediately re-compiled and successfully executed on new platforms. This relieves the programmer from maintaining large amounts of different, conditionally compiled (#ifdef'ed) code for the various platforms. It has the added advantage of providing the programmer with a clean, object-oriented C++ API to the OS services, instead of the native C API.

Files and Directories
Qt provides an API that allows Qt-based applications to query and manipulate the files and directories of the local file system in an OS-independent manner. Files and directories may be created, deleted, renamed, their access rights may be queried and modified, etc. The programmer is relieved from having to relate to such platform-specific details as, for example, the directory separator character in paths is "/" on Unix systems, and "\" on Windows.

Times and Dates
Qt provides classes for querying the system date and time. Dates and times may be operated with millisecond resolution. The time span between two different dates/times can be computed. Conversions to and from various date formats (Gregorian, Julian, seconds since the 1.1.1970 epoch, etc.) are provided. (Naturally, Qt's time and date handling is Y2K safe.)

Low-level I/O
Qt provides an API for OS-independent file I/O. The file I/O class is a specialization of Qt's general I/O device encapsulation class. It provides low-level I/O, i.e. reading and writing of raw blocks of bytes. Another specialization class provides I/O to a memory area. Custom encapsulations of other I/O devices may be added in the same way. Thus, an application may use the same code for doing I/O to files, memory buffers, and custom devices.

High-level I/O
Qt provides OS-independent, high-level, stream-based I/O. Both binary and text streams are provided. The streams use Qt's low-level I/O system, so they may be read from and written to files, memory buffers, and custom devices.

All the fundamental types (various precisions of integers and floating point values) and text strings may be read and written. The stream format is independent of the OS and the CPU byte-ordering (endianess), so the streams written on one OS / architecture may be read on any other.

Most of Qt's non-widget classes provide functionality for serializing their data to and from a binary stream, so they can efficiently be stored for later retrieval.

The text stream can be set to use a specific encoding / codec in order to read or write text in a format compatible with non-Qt applications.

8.2. Text Classes

Qt provides two string classes. QString is a powerful string class for all kinds of text operations. It operates with 16-bit Unicode characters. But for non-international applications Qt provides the QCString class. Seamless integration with the traditional C "char*" string is provided through automatic conversions.

QString
QString uses sharing, meaning that copies of a QString object will share the same string data in memory. The application programmer need not be concerned about the data sharing; if the application modifies the contents of one of the copied objects, QString automatically makes a deep copy of the string data so that the contents of the other copies remain unchanged. Sharing saves much memory and unnecessary copying.

QString provides all the usual string class functionality, such as searching, replacing, conversion to and from integer / floating-point values and various textual representations, comparison operators, truncation, insertion, etc. It automatically allocates enough memory space for the contents, so the programmer need not be bothered with this.

Qt provides a regular expression class for advanced text searching. Strings can be matched against regular expressions; the position and length of the match are returned, so it is straightforward to implement e.g. regular expression search and replace functionality.

QCString
For easy manipulation of non-internationalized text and classic C strings (where the conversion to and from QString's 16-bit representation could become an performance issue), Qt offers the QCString class, which provides most of the same functionality as QString.

8.3. Collection Classes

Qt includes a full set of generic, template-based collection classes. These allow the programmer to easily make, e.g., a stack class that operates on any Qt or programmer-defined class. The major collection classes provided are as follows:

- Array -- provides an ordered list of objects with constant-time indexed access.
- Dictionary -- stores a key value along with each object and provides fast (hashed) lookup based on the key values.
- Cache -- a Dictionary with a programmer-defined limit to the total number and/or cost of stored objects. When the limit is exceeded, the least recently accessed objects are discarded from the collection.
- Map -- a sorted list stored in a tree structure for efficient searching.
- List -- provides a double-linked list. For convenience, specialized List classes are provided for commonly used collection types, e.g. Sorted List and String List.

- Queue -- a first-in, first-out List.
- Stack -- a last-in, first-out List.

Corresponding Iterator classes are provided for all collection classes. The Iterators allow traversal of the entire collection independently of the collection's normal access method.

8.4. Network Classes

Qt offers a set of classes with cross-platform support for non-blocking, socket-based programming. The most important of these classes are:

QSocket
This class can be employed to establish a non-blocking socket for TCP, UDP (a UNIX-domain socket) or any other protocol family your operating system supports. It supports both binary and ASCII mode and offers a rich API for communicating state and accessing its content.

QSocketNotifier
Once you have created a QSocet you can create a QSocketNotifier to monitor the socket. You then connect the activated() signal to the slot you want to be called when a socket event occurs. There are three types of socket notifiers, the type is specified when the activated() signal is emitted.

- QSocketNotifier::Read: There is data to be read (socket read event).
- QSocketNotifier::Write: Data can be written (socket write event).
- QSocketNotifier::Exception: An exception has occured (socket exception event).

For example, if you need to monitor both reads and writes for the same socket, you must create two socket notifiers. Socket action is detected in the QApplication::exec() main event of Qt.

QServerSocket
This class is a convenience class for accepting incoming TCP connections. You can specify port or have QSocketServer pick one, listen on just one address or on all the addresses of a machine.

QDns
Both Windows and UNIX provide synchronous DNS lookups. Windows provides some asynchronous support too, but neither OS provides asynchronous support for anything other than hostname-to-address mapping.

QDns rectifies that by providing asynchronous caching lookups for the record types that we expect modern GUI applications to need in the near future. The aim of QDns is to provide a correct and small API to the DNS.

8.5. Threading

New in version 2.2, Qt offers cross-platform threading support.

QThread
A QThread represents a separate thread of control within the program. It shares all data with other threads within the process but executes independently in the way that a separate program does on a multitasking operating system. Each thread starts executing in QThread::run() and then dies whenever that function returns.

QMutex
The purpose of a QMutex is to protect an object, data structure or section of code so that only one thread can access it at a time. In Java terms, this is similar to the *synchronized* keyword.

9. Appendix 1: Widget Set

The most important widget classes provided in Qt are as follow:

QButtonGroup	Allows placing groups of button widgets together with a frame around and a header text. Typically used for logical grouping of radio buttons and check boxes.
QCanvasView	Displays a view of a QCanvas with scrollbars available, as for all ScrollView subclasses. There can be more than one view of a canvas.
QCheckBox	A button for displaying a nonexclusive switch, with an explanatory label. The label may be a text or a pixmap. Supports both binary on/off mode and tri-state on/grayed-out/off mode.
QComboBox	Allows selection of one from of a set of items which may be simple texts or pixmaps. Only the currently selected item is ordinarily displayed; the set of items is displayed in a pop-up menu. The user may optionally enter new text items by editing in the current item field.
QDialog	For building dialogs. Provides both modal and non-modal dialog semantics.
QIconView	A sophisticated new widget similar to QListView and QListBox. An iconview contains optionally labelled pixmap items that the user can select, drag around, rename, delete and more. The widget is highly optimized for speed and large amounts of icons.
QHeader	Provides column headers for tabular data displays. The user can drag the column separators to change the column width.
QLabel	For displaying static information (in the sense that the user cannot interact with the widget). The data can be a simple text string, a rich text, a pixmap, or a movie.
QLCDNumber	Displays numeric or restricted textual data in LCD panel style.
QLineEdit	Provides display and user editing of a single line of simple text. Supports native window system cut-and-paste and drag-and-drop.
QListBox	Allows selection of one or optionally several items from an item set. The items may be simple texts, pixmaps, or custom items (implemented as a subclass of QListBoxItem that takes care of the drawing of the item). Ordinarily, all items are displayed, unless they are too many or too wide to fit in the widget's available space, in which case scroll bars are provided automatically.
QListView	For display and user navigation in tree-structured lists in the style of, e.g., Windows Explorer. Both "Directory Tree" and "Directory List" display styles are supported. The user may expand and collapse branches. User selection of one or optionally several items is supported. Two types of items are provided by default: QListViewItem accepts a set of simple text strings where each string is displayed in a separate column; QCheckListItem accepts a text string and displays it with a radio button or a check box to allow the user to tick off any number of items. Custom item types may be added by sub-classing QListViewItem or QCheckListItem.
QMainWindow	A top level application window in "office suite" style. Supports a menu bar, tool bars, and a status bar, all of which may be turned on and off at run-time. The tool bars may be docked at any side of the window. Tool tips and What's This? help may also be added.
QMenuBar	Displays a menu bar at the top of a window. Menu bar items are QPopupMenu objects and may be presented as simple text strings, pixmaps, or a combination. Keyboard accelerators are supported.
QMultiLineEdit	Provides display and user editing of a multiple lines of simple text. Supports native window system cut-and-paste and drag-and-drop.
QPopupMenu	For displaying a pop-up or pull-down menu. Typically used in menu bars or as the right-mouse-button menu over other widgets. Items may be simple text

	strings, pixmaps, or a combination. Keyboard accelerators are supported. Checking (on/off) of menu bar items is optionally supported.
QProgressBar	Displays visual feedback on the progress of a lengthy operation, e.g. network downloading of large amounts of data.
QPushButton	The basic button. The button label may be a simple text string or a pixmap. Supports both normal single-shot mode and toggle (click-on/click-off) mode.
QRadioButton	A button for displaying an exclusive option, with an explanatory label. The label may be a text or a pixmap. Supports both binary on/off mode and tri-state on/grayed-out/off mode.
QScrollBar	For letting the user scroll the contents of other widgets when the contents are too large to fit in the available area. Both horizontal and vertical scroll bars are supported.
QScrollView	For building data display widgets that display just a part of a potentially very large virtual canvas.
QSlider	Lets the user specify a numeric value by dragging a caret along a groove. Vertical and horizontal modes are supported.
QSpinBox	Lets the user specify a numeric value either stepping use of the up- and down-buttons, or by entering it directly in the value field. Optional textual prefix and/or suffix are supported.
QSplitter	Splits an area between two or more widgets with dividing lines. The splits may be horizontal or vertical. Allows the user to drag the dividing lines to change the ratio of the area allocated to each widget.
QStatusBar	A message area typically used at the bottom of the main window in office-style applications. Supports both temporary and permanent messages.
QTabBar	A row of tabs for letting the user select which of a set of virtual pages to display. Supports both rounded and trapeze tab look, and looks suitable for placing both above and below the virtual pages.
QTableView	Creates tabular (spreadsheet style) data display widgets.
QTabWidget	Contains a stack of one or more virtual pages (i.e. programmer-provided widgets), and lets the user select which one should be displayed by selecting the corresponding tab.
QTextBrowser	Displays a rich text. Automatically provides scroll bars, as needed. Supports basic hypertext navigation facilities (forward, back, home) and anchors.
QTextView	Displays rich text, i.e. text containing XML-style formatting.
QToolBar	Provides a tool bar, typically used for short-hand access to frequently used functions in office-style applications.
QToolButton	A button designed to be used in tool bars. Supports text and/or icon label.
QToolTip	Gives the user pop-up tool tips (balloon help). Allows tool tip texts to be registered for any widget or part of a widget (static or dynamic). The widget's tool tip text gets displayed when the user lets the mouse cursor rest on a widget for a certain time.
QWhatsThis	Gives the user "What's This?" help. Allows help texts to be registered for any widget. When started, the What's This help changes the mouse cursor to a question mark and displays the help text for the widget that the user clicked on.
QWorkspace	Provides a workspace that can contain decorated windows, as opposed to frameless child widgets. QWorkspace makes it easy to implement a multidocument interface (MDI).

10. Ready-made Dialogs

Qt provides a number of ready-made dialog widgets for common tasks.

QColorDialog	Lets the user select a color either by dragging a cursor around on a spectrum area or by entering RGB or HSV values directly. Provides 48 predefined basic colors and up to 16 user-defined custom colors for quick selection.
QInputDialog	Convenient dialog used to get some simple input values from the user.
QFileDialog	Lets the user select a file or directory. Optionally allows multiple selections. Provides convenience functions for “Open” (single or multiple), “Save As”, and “Find Directory” dialogs. Supports file filters, e.g. “All C++ Files (*.cpp)”.
QFontDialog	Lets the user select a font. All fonts provided by the underlying window system are available for selection. From option lists the user may select the font name, style (bold/italic/underline/strikeout), size, and script (character set). A sample display of the currently selected font is provided.
QMessageBox	A modal dialog that displays an icon, a text and up to three push buttons. It's used for simple messages and questions
QTabDialog	Creates “Preferences...” style dialogs. Provides a QTabWidget, an “OK” push button, and optional “Apply”, “Cancel”, “Defaults”, and “Help” buttons. The button labels may be customized.
QWizard	Creates “Wizard” style dialogs, i.e. a dialog for leading the user through a process consisting of a number of steps, e.g. a software installation process. Each step is presented as a separate page, i.e. a programmer-provided widget. Provides “Back”, “Next”, “Finish”, “Cancel” and “Help” push buttons, as appropriate.

11. Appendix 2: Complete API Class List

*Part of the Qt OpenGL Module, the Qt Xt/Motif Extension or the Qt Netscape Plugin Extension

QAccel
QApplication
QArray
QAsciiCache
QAsciiCacheIterator
QAsciiDict
QAsciiDictIterator
QAsyncIO
QBig5Codec
QBitArray
QBitmap
QBitVal
QBoxLayout
QBrush
QBuffer
QButton
QButtonGroup
QCache
QCacheIterator
QCDEStyle
QChar
QCheckBox
QCheckListItem
QChildEvent
QClipboard
QCloseEvent
QCollection
QColor
QColorDialog
QColorGroup
QComboBox
QCommonStyle
QConnection
QConstString
QCString
QCursor
QCustomEvent
QCustomMenuItem
QDataPump
QDataSink
QDataSource
QDataStream
QDate
QDateTime
QDialog
QDict
QDictIterator
QDir
QDoubleValidator
QDragEnterEvent
QDragLeaveEvent
QDragMoveEvent
QDragObject
QDropEvent
QDropSite
QEvent
QFile
QFileDialog
QFileIconProvider
QFileInfo
QFocusData
QFocusEvent
QFont
QFontDataBase
QFontDialog
QFontInfo
QFontMetrics
QFrame
QGArray
QGCache
QGCacheIterator
QGDict
QGDictIterator
QGL*
QGLayoutIterator
QGLContext*
QGLFormat*
QGList
QGListIterator
QGLWidget*
QGrid
QGridLayout
QGroupBox
QGuardedPtr
QHBox
QHBoxLayout
QHButtonGroup
QHeader
QHGroupBox
QHideEvent
QIconSet
QIconView
QImage
QImageConsumer
QImageDecoder
QImageDrag
QImageFormat
QImageFormatType
QImageIO
QInputDialog
QIntCache
QIntCacheIterator
QIntDict
QIntDictIterator
QIntValidator
QIODevice
QIODeviceSource
QKeyEvent
QLabel
QLayout
QLayoutItem
QLayoutIterator
QLCDNumber
QLineEdit
QList
QListBox
QListBoxItem
QListBoxPixmap
QListBoxText
QListIterator
QListView
QListViewItem
QListViewItemIterator
QLNode
QMainWindow
QMap
QMapConstIterator
QMapIterator
QMenuBar
QMenuData
QMessageBox
QMetaProperty
QMimeSource
QMimeSourceFactory
QMotifStyle
QMouseEvent
QMoveEvent
QMovie
QMultiLineEdit
QNetworkProtocol
QNPInstance*
QNPlugin*
QNPStream*
QNPWidget*
QObject
QPaintDevice
QPaintDeviceMetrics
QPainter
QPaintEvent
QPalette

QPen
QPicture
QPixmap
QPixmapCache
QPlatinumStyle
QPNGImagePacker
QPoint
QPointArray
QPopupMenu
QPrinter
QProgressBar
QProgressDialog
QPtrDict
QPtrDictIterator
QPushButton
QQueue
QRadioButton
QRangeControl
QRect
QRegExp
QRegion
QResizeEvent
QScrollBar
QScrollView
QSemiModal
QSessionManager
QShared
QShowEvent
QSignal
QSignalMapper
QSimpleRichText
QSize
QSizeGrip
QSizePolicy
QSlider
QSocketNotifier
QSortedList
QSpacerItem
QSpinBox
QSplitter
QStack
QStatusBar
QStoredDrag
QStrIList
QString
QStringList
QStrList
QStyle
QStyleSheet
QStyleSheetItem
Qt
QTab
QTabBar
QTabDialog
QTableView
QTabWidget
QTextBrowser
QTextCodec
QTextDecoder
QTextDrag
QTextEncoder
QTextIStream
QTextOStream
QTextStream
QTextView
QTime
QTimer
QTimerEvent
QToolBar
QToolButton
QToolTip
QToolTipGroup
QTranslator
QUriDrag
QUrl
QUrlOperator
QValidator
QValueList
QValueListConstIterator
QValueListIterator
QValueStack
QVariant
QVBox
QVBoxLayout
QVButtonGroup
QVGroupBox
QWhatsThis
QWheelEvent
QWidget
QWidgetItem
QWidgetStack
QWindowsStyle
QWizard
QWMatrix
QWorkspace
QXtApplication*
QXtWidget*