Qt

Cross-platform C++ GUI Application Framework

Technical Overview

Contents

Introduction

Software producers have for many years had to face the problem of how to target a market consisting of a diversity of operating systems and window systems. There are no indications that this situation is about to change. It is now seen that the early nineties' rumors of the impending death of Unix were very much exaggerated. The current boom of Linux, and its positioning as a competitor on the desktop, makes it clear that the software market will continue to consist of many different platforms in the foreseeable future.

A major challenge in targeting multiple platforms is the cost of developing and maintaining an application for several different platforms. Because of the inherent differences between the platforms, the porting of an application to a new platform will in practice often involve redesign and re-implementation.

This white paper presents Qt, a software development application framework that solves many of the greatest challenges of cross-platform application development and maintenance. It explains the principles of how software developers can use Qt to create single code base applications for end-users on different platforms.

Qt is a product of Trolltech. It has been on the market since 1995, and is used by leading companies like HP, IBM, Intel, Siemens, Xerox, etc. Qt is particularly widely used on Linux, where it forms the basis of the popular KDE desktop environment.

A detailed presentation of all the functionality provided by Qt is outside the scope of this document. Further technical information is available in:

- The Qt Reference Documentation. Available on-line at www.trolltech.com/qt.
- *Programming with Qt*, by Matthias Kalle Dalheimer. O'Reilly, 1999.

Qt is a continuously evolving toolkit. This document presents the main features of the current version of Qt. At the time of writing, this is version 2.0.

Further information about Qt is available at the Trolltech web site:

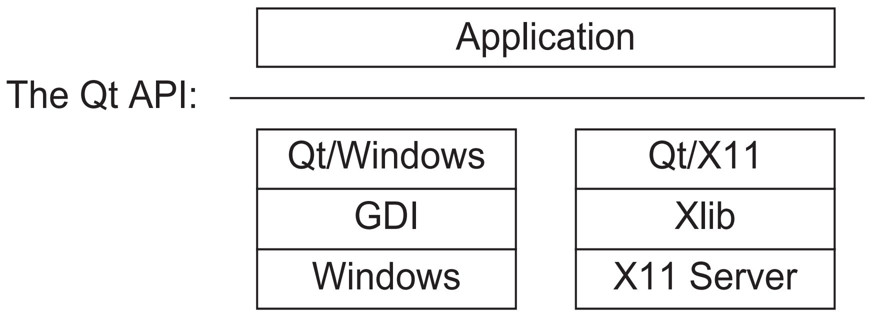
www.trolltech.com

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) to application developers. Qt provides a wide spectrum of generally useful functionality, but the main focus is on the GUI (Graphical User Interface). Thus, for application developers Qt replaces Motif, MFC, and/or other GUI toolkits.

Cross-Platform Development

Qt is cross-platform, in the sense that the Qt class library is implemented for several different operating and window systems. 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 by simply recompiling it on the new platform, and linking it with the Qt library for that platform. 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 two main groups of operating systems:

- Unix: This covers 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 uses the X Window system.
- Windows: Covers Windows 95, 98, and NT. The Qt library is implemented using the Windows GDI API, and uses the Microsoft Windows window system.

Implementations for other operating and window systems are planned.

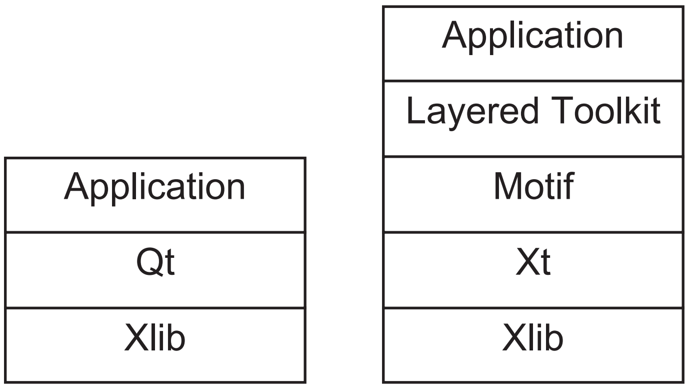
The Qt library code is designed to be extremely portable. All major hardware architectures for the various operating systems are supported, including 64 bit systems. The Unix / X11 implementation is successfully employed in commercial use on not only the operating systems mentioned above, but also on real-time operating systems like QNX and VxWorks. It is also being used on OS/2 Warp using the XFree86 X server.

Layered versus Emulating Architectures

When designing a cross-platform library like Qt, one can take two fundamentally different approaches. Either, one can choose to build a wrapper API on top of the native GUI components (widgets), i.e. MFC on Windows and Motif on Unix – a *layered* architecture. Or, one can choose to build a toolkit directly on top of the lowest level provided by the platforms, e.g. the GDI on Windows and Xlib on Unix. 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. Hence, this is called an *emulating* architecture.

This kind of emulating libraries should not be confused with the libraries that emulate one native API on top of another, e.g. an MFC emulator for Unix / X11. This is a very different architecture, and will be discussed later.

In the cross-platform GUI library industry, the pros and cons of the two types of architecture has been discussed for years. Initially, the layered architecture was generally perceived to be preferable. Hence, when this industry boomed for the first time in the late eighties, most products were layered. However, the nineties have seen many of these products fail, being no longer maintained nor supported.



Qt is designed with an emulating architecture. In the following, the main issues in the layered vs. emulating architecture discussion will be presented, together with the reasons why the emulating solution was chosen for Qt.

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. An emulating toolkit must, as the name implies, emulate the native GUI elements, and this emulation will unavoidably be imperfect. The strength of this argument builds 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.

However, though these assumptions were relevant 5 to 10 years ago, the situation is now different. At that time, there were many contenders for the position as *the* standard GUI: Windows, OS/2 Presentation Manager, Macintosh, Motif, and others. The ensuing "religious wars" made users extremely sensitive on look and feel issues. Today, this battle is no longer the order of the day, and with the introduction of Java and other technologies, a diffusion of the strict look and feel standards has taken place. 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 would have been a difficult task indeed. However, in the later years the pace of native look and feel development has slowed down considerably. For example, no Windows version since Windows 95 has introduced anything more than minor extensions and variations, as mentioned above. It seems highly probable that this trend will continue, because of the sheer number of users that 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 well up-to-date.

When considering the look and feel argument, it should also be noted that even a layered toolkit will be forced to resort to emulation if it wants to provide GUI elements that are not offered by the native GUI. The alternative is to be a so-called least common denominator toolkit, which is not a satisfactory solution. Furthermore, custom 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 (slightly) 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.

In addition, an API is provided for implementing custom styles. This means that for applications that have special demands regarding visual appearance, e.g. a kiosk application, it is straightforward for the programmer to implement a custom look and feel. All visual elements in Qt will then present themselves using this custom 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 custom styles.

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 will be excessively bulky since it must include not only the toolkit itself, but also all the layers below. Proponents of the layered approach will on the other hand 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).

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.

Native-API Emulation Libraries

Another possible design for a cross-platform library is to emulate the native API of one platform on another. This is the architecture chosen by for example the Unix / X11 MFC emulation libraries. The goal of this design is to allow a legacy application (built using the native API) to be quickly made to run on another platform, by avoiding the need to re-implement it using the special API of the cross-platform toolkit.

Qt does not use this design, for several reasons:

- It is really a special kind of layered design, with all the disadvantages of that architecture discussed above. The performance problem is likely to be particularly serious, because yet another layer is introduced.
- Application programmers often find the existing native APIs, such as Motif and MFC, to be complex and cumbersome to use. Qt is designed to offer the 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, they bypass the native API to achieve special effects by accessing the lower layers directly, etc. Attempting to emulate the native API close enough to handle such code is extremely difficult.

Performance

Run-time efficiency and performance is a central design goal of the 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 in such cases where experiments have determined that the latter is slower.

One of the techniques used in Qt for improving 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 it reduces the memory demands of the application as a whole. This technique is especially effective when applied to classes that contain large data amounts, such as pixmaps and images, and when applied to classes that are frequently used, such as strings. All such classes in Qt use sharing.

API design

Some important design goals of Qt are:

- Effective use of object-oriented principles. For example, all widget classes (both ready-made and custom) 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 the supported window systems, by implementing them internally for platforms lacking them. 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 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.

Signals and Slots

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 offers assistance to the application programmers for this task, in the form of a special inter-object communication mechanism called *signals and slots*. It 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 discussed below.

The signal / slot mechanism consists of the following constructs:

- A class may define any of its (otherwise normal) member functions to be *slots*.
- A 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 object may be *connected* to a slot of another object.
- An object may at any time choose to *emit* a signal.

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

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 either 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 handled gracefully. Such mismatches in callback functions invariably lead to run-time failures (segmentation faults) and hard application termination.

The typical use of the signal-slot mechanism is best illustrated by an example. Assume an application design calls for a dialog box that gets closed when the user clicks its "OK" button. Using Qt, the programmer will implement this using 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.

Internationalization

Qt is designed to allow applications to use any language and character set.

Unicode

Qt allows applications to use international (i.e. non-ASCII) character sets. For text operations, 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, and it is used in all API functions that take or return text parameters. This includes all text labels of widgets, e.g. the labels on push buttons, menu items, etc.

A note about performance: QString is highly optimized, and in our tests of moving real-world applications from 8 bit to 16 bit strings, no significant performance penalty was observed. This is to be expected, since text manipulation is not among the most demanding tasks performed by typical GUI applications.

Qt supports keyboard input and screen output of Unicode text, as 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. Custom 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.

Localization

Qt provides support for creating *localized* applications, i.e. applications that can choose at run-time what language to display all the user-visible texts in. The choice may be made automatically based on the user's locale setting, or explicitly by the application (e.g. by presenting the user with a language selection dialog on start-up).

Building a Qt-application that is prepared for localization is straightforward: 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 function tr() will do a lookup in the currently selected translation table, and return the text string (translation) corresponding to the argument.

A handy aspect of the tr() function is that if no translation table is installed, it will simply return its argument. This means that a localization-prepared application will run just fine even if the translation tables are not present; its behavior will be the same as if it were not localization-prepared. This is practical during application development (when translation tables have not been produced yet), or for releasing the first version(s) of an application which is planned to be localized in later versions.

Qt provides tools which assist the application developers in building and maintaining the translation tables. One tool 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 converts these text files to the binary, hashed translation table files that are used by Qt for lookup at run-time. A third tool assists in merging existing translation files when the application has been extended or modified so that new strings that need translation have been added.

Graphical User Interfaces

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, a push button is created by making 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* are the basic elements like buttons and scroll bars, while *containers* are the elements that contain the controls, like dialogs and application windows. Qt is more flexible: there is no fundamental difference between containers and controls; any widget may function as the one or the other. Containment is expressed in a *parent-child* relationship: A widget that contains other widgets is called the *parent* of the contained widgets.

A widget class provides an API to access the contents 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; e.g. 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. either by mouse click, by keyboard accelerator, or programmatically from another part of the application. This makes it easier to ensure internal consistency in the application.

User Interface Composition

Making a normal application window is straightforward. The application programmer starts by creating an object of a suitable container widget class. Then, the various controls are 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 taken care of by a layout manager (more about this 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 to build user interfaces from. This includes classes for all the common GUI controls typically found in modern user interfaces, such as buttons, scroll bars, tool bars, explorer-style hierarchical list views, etc. Thus, normal user interfaces can be constructed rapidly by composing standard widgets as described above. A complete listing of the standard widget classes is given in Appendix 1.

Custom Widget Classes

A central design feature of Qt's widget system is *extensibility*. This is important, since experience shows that a fixed set of static widget classes cannot cover all the requirements of a real-world application. GUI toolkit designers can try to foresee the various demands that applications may have, and try to provide the necessary functionality in the widget classes (indeed, Qt's standard widget classes are designed like this), but that can never be a satisfactory replacement for enabling the application developers to easily customize the widget classes or to design their own widget classes from scratch.

Qt is designed to make it very easy for the application programmers to create custom 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, etc.

Custom widgets have several powerful uses:

Application Data Presentation

Custom widget classes can be used to implement applications' fundamental and unique graphical interfaces, e.g. 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 custom widget class will employ Qt's graphics API for the presentation, and the event system to implement user navigation and data manipulation.

Qt provides some widget classes that are particularly suited for use as the basis for implementing these kinds of widgets:

- QScrollView provides subclasses with a framework for building widget classes that display just a part of a potentially much larger virtual canvas area. Scroll bars are automatically provided as necessary. The contents may consist of directly drawn graphics 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.
- QTableView provides a framework for building widgets that display data in tabular (spread sheet style) format.

Tuning the Behavior of Standard Controls

Often, an application needs a GUI control that is almost, but not quite, the same as one of the standard controls. 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, thus overriding the original implementation.

Custom Controls

Sometimes, applications will require some new kinds of basic user interaction elements. For example, a word processor may require a ruler for letting the user specify the tab stops in a word processor. By creating a custom widget class, the application programmers can achieve precisely the desired behavior. All the functionality of Qt that is used to implement its standard widget classes is also available to the custom widget class programmers.

Naturally, 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.

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 still keeping the window contents. This will fail 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 in advance know 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 impression. Maintenance is also demanding, since the whole layout must be manually re-implemented whenever widgets are added or removed.

To let Qt-based applications overcome all the above problems, Qt provides a mechanism for automatic widget layout management. An API is provided so that a widget may create a *layout manager* object, which will then take care of assigning positions and sizes to 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 gets resized, or a child widget's size requirements change, the layout manager will automatically recalculate the layout, and move and resize all the child widgets to fit.

Qt provides two basic layout manager classes (custom layout manager classes may also be added):

- 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.

Instead of a child widget, a cell may contain another layout manager object, which in turn manages other child widgets. Thus, by building a nested structure of layout managers, automatic layout of even very complex user interfaces can be readily achieved.

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 to not 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 will for all of these constraints provide sensible default values, calculated at run-time depending on the widgets' current contents. If the contents of a widget change while the program is running, the layout will be automatically recalculated to fit the new size of the widget.

The layout algorithm may be tuned as follows:

- A stretch factor can be 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 can be 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 may be specified.
- The maximum and/or minimum size of the child widget may be set explicitly.

Graphics

Qt provides much 2D graphics functionality. The basic graphics API in Qt is the QPainter class. This class provides a high-level drawing engine with commands for drawing lines, polygons, ellipses, splines, images, etc.

Device Independent Graphics

Qt supports graphics drawing to screen (i.e. widgets), printers, pixmaps and pictures (known as a meta-file in Windows terminology). The intrinsic differences between these devices are hidden 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 example may applications use the same drawing routine for screen output as for print output. This is done by simply making the drawing routine take the QPainter as a parameter, and then passing one opened on a widget for screen drawing, and one opened on a printer for printer drawing.

Special Paint Devices

In addition to widgets and printers, Qt supports drawing to the following special devices (as always, through a QPainter):

QPixmap

A pixmap is an off-screen memory frame area, i.e. a two-dimensional array of pixel values. If an application is going to display complex static graphics on screen, it makes sense to draw the graphics into a pixmap, and then just draw the pixmap to 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. Qt provides fast bitblt (bit block transfer) operations for moving pixels between widgets and pixmaps.

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

QPicture

A picture is a stored sequence of drawing operations. Pictures are very handy for storing graphics for re-display at a different magnification level, for instance. 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.

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 lacking in the underlying system (e.g. drawing transformations on Windows 95/98 and X11) are implemented in Qt itself.

Color handling

Qt provides a separate class for specifying color for drawing operations. Colors can be 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 handles the allocation of colors in the system palette, so Qt-based programs need not do anything special to be prepared for running on such systems.

The Drawing Style

For specifying the desired graphics attributes of lines, polygon outlines, etc., Qt provides the QPen class. The color, line width, and stipple pattern can be set.

For the filling style of polygons, ellipses etc., Qt provides the QBrush class. With this, the fill color and fill pattern can be specified. A set of predefined patterns are available; a custom fill pattern (specified as a pixmap) can also be set.

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 e.g. the drawRect() method) 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.

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

For the view transform, Qt allows setting the origin and extent of the drawing window and the drawing viewport. The drawing viewport determines the logical coordinate system, and specifying this to e.g. 1000 x 1000 gives the application programmer a 1000 x 1000 drawing area independently 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 Qt should 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 for specifying the font. All the fonts installed in the underlying window system may be used for text drawing 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, and Japanese and various other Asian character sets.

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 image formats are supported on all platforms, 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, and also allows application programmers to 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 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 bits 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.

3D Graphics

Qt does not itself offer 3D graphics functionality, but integration with 3rd party 3D libraries is provided.

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. Thus, it is possible for application programmers to build custom integration packages to other libraries.

OpenGL

Integration with OpenGL is provided by the Qt OpenGL Extension library. It allows the 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 offers a HOOPS-Qt integration package.

Tool Classes

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

Some of Qt's tool classes provide similar functionality to the C++ standard library and the STL. However, Qt's tool classes are preferable for most Qt-based applications, for the following reasons:

- Portability: The 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 the Qt classes, the application programmers are relieved from relating to 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 the 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 the case with the standard classes.

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

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 the 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 that the directory separator character in paths is "/" on Unix systems, and "\" on Windows.

Times and Dates

Classes for querying the system date and time are provided. Dates and times may be operated on with millisecond resolution. The time span between two different dates/times can be computed. Conversion 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.

Text Classes

Qt provides a powerful string class for all kinds of text operations. It operates with 16 bit Unicode characters, but for non-international applications, it provides seamless integration with the traditional C "char*" string through automatic conversions.

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, like 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 is relieved from managing this.

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

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

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. These are the major collection classes provided:

- 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.

For all collection classes, corresponding Iterator classes are provided. The Iterators allow traversal of the entire collection independently of the collection’s normal access method.

Appendix 1: Widget Set

This is a list of the most important widget classes provided in Qt.

QButtonGroup	For 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.
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.
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 automatically provided.
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 QPopup-Menu 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 is 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 supported.
QSpinBox	Lets the user specify a numeric value either stepping using 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.
QTabDialog	Used for creating “Preferences...” style dialogs. Provides a QTabWidget, an “OK” push button, and optional “Apply”, “Cancel”, “Defaults”, and “Help” buttons. The button labels may be customized.
QTableView	For creating 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	For giving 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). When the user lets the mouse cursor rest on a widget for a certain time, the widget's tool tip text gets displayed.
QWhatsThis	For giving the user "What's This?" help. Allows help texts to be registered for any widget. When started, the What's This help will change the mouse cursor to a question mark, and will display the help text for the widget the user clicks on.
QWizard	Used for creating "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.

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.
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. By selecting 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.

Appendix 2: Complete API Class List

QAccel	QDropEvent	QLabel	QProgressDialog	QTextOStream
QApplication	QDropSite	QLayout	QPtrDict	QTextStream
QArray	QEvent	QLayoutItem	QPtrDictIterator	QTextView
QAsciiCache	QFile	QLayoutIterator	QPushButton	QTime
QAsciiCacheIterator	QFileDialog	QLCDNumber	QQueue	QTimer
QAsciiDict	QFileIconProvider	QLineEdit	QRadioButton	QTimerEvent
QAsciiDictIterator	QFileInfo	QList	QRangeControl	QToolBar
QAsyncIO	QFocusData	QListBox	QRect	QToolButton
QBitArray	QFocusEvent	QListBoxItem	QRegExp	QToolTip
QBitmap	QFont	QListBoxPixmap	QRegion	QToolTipGroup
QBitVal	QFontDialog	QListBoxText	QResizeEvent	QTranslator
QBoxLayout	QFontInfo	QListIterator	QScrollBar	QUriDrag
QBrush	QFontMetrics	QListView	QScrollView	QValidator
QBuffer	QFrame	QListViewItem	QSemiModal	QValueList
QButton	QGArray	QListViewItemIterator	QSessionManager	QValueListConstIt-
QButtonGroup	QGCache	QLNode	QShared	erator
QCache	QGCacheIterator	QMainWindow	QShowEvent	QValueListIterator
QCacheIterator	QGDict	QMap	QSignal	QVBox
QCDEStyle	QGDictIterator	QMapConstIterator	QSignalMapper	QVBoxLayout
QChar	QGL*	QMapIterator	QSimpleRichText	QVButtonGroup
QCheckBox	QGLayoutIterator	QMenuBar	QSize	QVGroupBox
QCheckListItem	QGLContext*	QMenuData	QSizeGrip	QWhatsThis
QChildEvent	QGLFormat*	QMessageBox	QSizePolicy	QWheelEvent
QClipboard	QGList	QMimeSource	QSlider	QWidget
QCloseEvent	QGListIterator	QMimeSourceFactory	QSocketNotifier	QWidgetItem
QCollection	QGLWidget*	QMotifStyle	QSortedList	QWidgetStack
QColor	QGrid	QMouseEvent	QSpacerItem	QWindowsStyle
QColorDialog	QGridLayout	QMoveEvent	QSpinBox	QWizard
QColorGroup	QGroupBox	QMovie	QSplitter	QWMatrix
QComboBox	QHBox	QMultiLineEdit	QStack	QXtApplication*
QCommonStyle	QHBoxLayout	QNPInstance*	QStatusBar	QXtWidget*
QConnection	QHButtonGroup	QNPlugin*	QStoredDrag	
QConstString	QHeader	QNPStream*	QStrIList	
QCString	QHGroupBox	QNPWidget*	QString	
QCursor	QHideEvent	QObject	QStringList	
QCustomEvent	QIconSet	QPaintDevice	QStrList	
QDataPump	QImage	QPaintDeviceMetrics	QStyle	
QDataSink	QImageConsumer	QPainter	QStyleSheet	
QDataSource	QImageDecoder	QPaintEvent	QStyleSheetItem	
QDataStream	QImageDrag	QPalette	Qt	
QDate	QImageFormat	QPen	QTab	
QDateTime	QImageFormatType	QPicture	QTabBar	
QDialog	QImageIO	QPixmap	QTabDialog	
QDict	QIntCache	QPixmapCache	QTableView	
QDictIterator	QIntCacheIterator	QPlatinumStyle	QTabWidget	
QDir	QIntDict	QPNGImagePacker	QTextBrowser	
QDoubleValidator	QIntDictIterator	QPoint	QTextCodec	
QDragEnterEvent	QIntValidator	QPointArray	QTextDecoder	
QDragLeaveEvent	QIODevice	QPopupMenu	QTextDrag	
QDragMoveEvent	QIODeviceSource	QPrinter	QTextEncoder	
QDragObject	QKeyEvent	QProgressBar	QTextIStream	

**Part of the Qt OpenGL, Image I/O, or Motif/Xt Extensions*