

## **Qt 3.0 Whitepaper**

*Trolltech*

### **Abstract**

This whitepaper describes the Qt C++ toolkit. Qt supports the development of cross-platform GUI applications with its 'write once, compile anywhere' approach. Using a single source tree and a simple recompilation, applications can be written for Windows 95/98/NT4/ME/2000, Mac OS X, Linux, Solaris, HP-UX and many other versions of Unix with X11. Qt applications can also be compiled to run on Qt/Embedded. Qt introduces a unique inter-object communication mechanism called 'signals and slots'. Qt has excellent support for many programming domains: 2D and 3D graphics, internationalization, XML, etc. Qt applications can be built visually using *Qt Designer*.

# Qt 3.0 Whitepaper

*Trolltech*

## Contents

1. Introduction . . . . .	3
1.1. Executive Summary . . . . .	3
2. Widgets . . . . .	4
2.1. A 'Hello' Example . . . . .	5
2.2. Built-in Widgets . . . . .	5
2.3. Custom Widgets . . . . .	7
3. Signals and Slots . . . . .	8
3.1. A Signals and Slots Example . . . . .	9
3.2. Meta Object Compiler . . . . .	10
4. GUI Applications . . . . .	11
4.1. Main Window Classes . . . . .	11
4.2. Multiple Document Interface . . . . .	14
4.3. Dialogs . . . . .	14
4.4. Dock Windows . . . . .	16
4.5. Settings . . . . .	17
4.6. Multi-Threading . . . . .	17
5. Qt Designer . . . . .	18
5.1. Qt Assistant . . . . .	19
5.2. GUI Application Example . . . . .	20
6. 2D and 3D Graphics . . . . .	22
6.1. 2D Graphics . . . . .	22
6.2. 3D Graphics . . . . .	26
6.3. A 3D Example . . . . .	27
7. Databases . . . . .	29
7.1. Executing SQL Commands . . . . .	30
7.2. Data-aware Widgets . . . . .	31
8. Internationalization . . . . .	32
8.1. Unicode . . . . .	32
8.2. Text Rendering . . . . .	32
8.3. Input Methods . . . . .	33
8.4. Dates and Times . . . . .	33
8.5. Translating Applications . . . . .	33
8.6. Qt Linguist . . . . .	34
9. Styles and Themes . . . . .	35
9.1. Built-in Styles . . . . .	35

9.2. Style-aware Widgets	35
9.3. Custom Styles	35
10. Layouts	36
10.1. Built-in Layout Managers	37
10.2. Nested Layouts	37
10.3. Custom Layouts	39
11. Events	39
11.1. Event Creation	39
11.2. Event Delivery	39
12. Input/Output and Networking	40
12.1. Input/Output	40
12.2. XML	41
12.3. Inter-Process Communication	41
12.4. Networking	42
13. Collection Classes	43
13.1. Value-based Collections	43
13.2. Pointer-based Collections	44
14. Plugins and Dynamic Libraries	44
14.1. Dynamic Libraries	44
14.2. Plugins	44
15. Qt's Architecture	45
15.1. Microsoft Windows	46
15.2. X11	46
15.3. Mac OS X	47
15.4. Embedded Linux	47
15.5. Qt's Development World	47
<b>Index</b>	<b>48</b>

# 1. Introduction

*Qt is a C++ toolkit for cross-platform GUI and application development. In addition to the Qt C++ class library, the toolkit includes tools to make writing applications fast and straightforward. Qt's cross-platform capabilities and extensive internationalization support ensure that Qt applications reach the widest possible market.*

The Qt C++ toolkit has been at the heart of commercial applications since 1995. Qt is used by companies as diverse as AT&T, IBM, NASA and Xerox, and by numerous smaller companies and organizations. Qt 3.0 retains the ease-of-use and power of earlier versions while adding significant functionality and introducing many new classes. Qt's classes are fully featured to reduce developer workload, and provide consistent interfaces to speed learning. Qt is, and always has been, fully object oriented.

This whitepaper gives an overview of Qt's tools and functionality. Each section begins with a non-technical introduction, then presents the technical details in increasing depth. Code extracts, and small complete applications, are presented. To evaluate Qt for 30 days, visit <http://www.trolltech.com>.

## 1.1. Executive Summary

Qt includes a rich set of widgets [p. 4] ('controls' in Windows terminology) that provide standard GUI functionality. Qt introduces an innovative alternative for inter-object communication, called 'signals and slots' [p. 8], that replaces the old and unsafe callback technique. Qt also provides a conventional events [p. 39] model for handling mouse clicks, key presses, etc. Qt's cross-platform GUI applications [p. 11] can use all the user interface functionality required by modern applications, such as menus, context menus, dockable toolbars, balloon help, drag and drop, etc.

Intuitive naming conventions and a consistent programming approach simplify coding. Qt also includes *Qt Designer* [p. 18], a tool for designing user interfaces graphically. *Qt Designer* supports Qt's powerful layouts [p. 36] in addition to absolute positioning. *Qt Designer* can be used purely as a design tool, or it can be used to create entire applications with the built-in C++ code editor.

Qt has excellent support for 2D and 3D graphics [p. 22]. Qt is the de-facto standard GUI toolkit for platform-independent OpenGL programming.

Qt makes it possible to create platform-independent database applications using standard databases [p. 29]. Qt includes native drivers for Oracle, Microsoft SQL Server, Sybase Adaptive Server, PostgreSQL, MySQL and ODBC-compliant databases. Qt's database functionality is fully integrated with *Qt Designer*, which offers live preview of database data. Qt includes database-specific widgets, and any built-in or custom widget can be made data aware.

Qt programs have native look and feel on all supported platforms using Qt's styles and themes support [p. 35]. From a single source tree, recompilation is all that is required to produce applications for Windows 95/98/NT4/ME/2000, Mac OS X, Linux, Solaris, HP-UX and many other versions of Unix with X11. Qt applications can also be compiled to run on Qt/Embedded. Qt's `qmake` build tool produces Makefiles or `.dsp` files appropriate to the target platform. Since Qt's architecture [p. 45] takes advantage of the underlying platform, many customers also use Qt for single-platform development on both Windows and on Unix because they prefer the Qt approach.

Qt uses Unicode throughout and has considerable support for internationalization [p. 32]. Qt includes *Qt Linguist* [p. 34] and other tools to support translators. Applications can easily use and mix text in Arabic, English, Hebrew, Japanese, Russian and all the other languages supported by Unicode.

Qt includes a variety of domain-specific classes. For example, Qt has an XML module [p. 41] that includes SAX and DOM parsers. Objects can be stored in memory using Qt's STL-compatible collection classes [p. 43]. Local and remote file handling using standard protocols are provided by Qt's input/output and networking classes [p. 40].

Qt applications can have their functionality extended by plugins and dynamic libraries [p. 44]. Plugins provide additional codecs, database drivers, image formats, styles and widgets. Libraries can offer an unlimited range of functionality. Plugins and libraries can be sold as products in their own right.

Qt is a mature, solid C++ toolkit that is widely used across the world. In addition to Qt's many commercial uses, the free edition of Qt is the foundation of KDE, the Linux desktop environment. Qt makes application development a pleasure, with its cross-platform build system, visual form design and elegant API.

### On-line References

<http://www.trolltech.com/references/customers/>  
<http://www.trolltech.com/references/partners/>

## 2. Widgets

*Qt has a rich set of widgets (buttons, scroll bars, etc.) that cater for most situations. Qt's widgets are flexible and easy to subclass for special requirements.*

Qt provides a full set of widgets. Widgets are visual elements that are combined to create user interfaces. Buttons, menus, scroll bars, message boxes, application windows, etc., are all examples of widgets. Qt's widgets are not arbitrarily divided between 'controls' and 'containers'; all widgets can be used both as controls and as containers. Custom widgets can easily be created by subclassing existing Qt widgets, or created from scratch on the rare occasion when this is necessary.

Widgets are instances of **QWidget** or one of its subclasses, and custom widgets are created by subclassing.

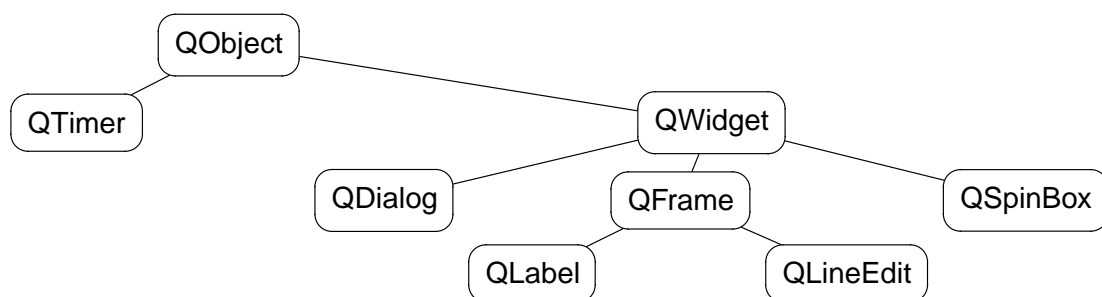


Figure 1. An extract from the **QWidget** class hierarchy

A widget may contain any number of child widgets. Child widgets are shown within the parent widget's area. A widget with no parent is a top-level widget (a 'window'), and usually has its own entry in the desktop environment's task bar. Qt imposes no arbitrary limitations on widgets. Any widget can be a top-level widget; any widget can be a child of any other widget. The position of child widgets within the parent's area can be set automatically using layout managers [p. 36], or manually if preferred.

When a parent widget is disabled, hidden or deleted, the same action is applied to all its child widgets recursively.

Labels, message boxes, tooltips, etc., are not confined to using a single color, font and language. Qt's text-rendering widgets can display multi-language rich text using a subset of HTML. See Text Rendering [p. 32].

## 2.1. A 'Hello' Example



Figure 2. Hello Qt!

The complete source code for a program that displays “Hello Qt!” follows:

```
#include <qapplication.h>
#include <qlabel.h>

int main( int argc, char **argv )
{
    QApplication app( argc, argv );
    QLabel *hello = new QLabel( "<font color=blue>Hello <i>Qt!</i>"
                               "</font>", 0 );
    app.setMainWidget( hello );
    hello->show();
    return app.exec();
}
```

## 2.2. Built-in Widgets

The screenshots below present the main Qt widgets. They are shown using the Windows style.

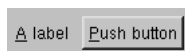


Figure 3. A **QLabel** and a **QPushButton** laid out with a **QHBoxLayout**

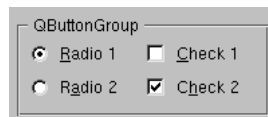


Figure 4. Two **QRadioButton**s and two **QCheckBox**s laid out with a **QButtonGroup**

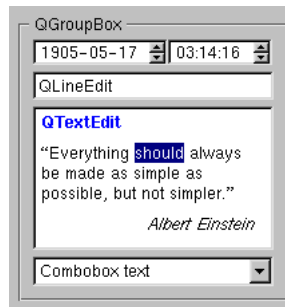


Figure 5. A **QDateEdit**, a **QLineEdit**, a **QTextEdit** and a **QComboBox** laid out with a **QGroupBox**

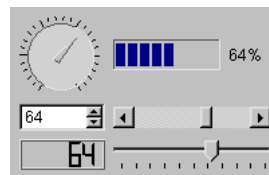


Figure 6. A **QDial**, a **QProgressBar**, a **QSpinBox**, a **QScrollBar**, a **QLCDNumber** and a **QSlider** laid out with a **QGrid**

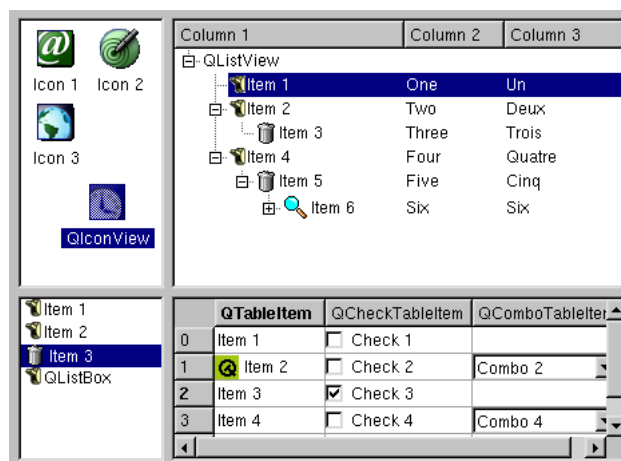


Figure 7. A **QIconView**, a **QListView**, a **QListBox** and a **QTable** laid out with a **QGrid**

**QComboBox**, **QLineEdit** and **QSpinBox**'s input can be constrained or validated using a **QValidator** subclass. Qt regular expressions can be used for validation.

Widgets that are used to display large amounts of data (e.g. **QTable**, **QListView** and **QTextEdit**) inherit **QScrollView** and can display scroll bars automatically.

**QMenuBar**, **QStatusBar**, **QToolBar** and related widgets are presented in GUI Applications [p. 11]. **QMessageBox**, **QFileDialog**, **QTabDialog**, **QWizard** and other dialogs are presented in Dialogs [p. 14]. **QSplitter** is covered in Layouts [p. 36]. **QCanvas** and **QGLWidget** are presented in 2D and 3D Graphics [p. 22].

The screenshot that shows the **QRadioButtons** and **QCheckBoxes** was produced with the following code:

```

parent = new QButtonGroup( 2, Qt::Vertical, "QButtonGroup" );
radio1 = new QRadioButton( "&Radio 1", parent );
radio2 = new QRadioButton( "R&adio 2", parent );
radio1->setChecked( TRUE );
check1 = new QCheckBox( "&Check 1", parent );
check2 = new QCheckBox( "C&heck 2", parent );
check2->setChecked( TRUE );

```

## 2.3. Custom Widgets

Developers can create their own widgets and dialogs by subclassing **QWidget** or one of its subclasses. To illustrate subclassing, the complete code for a digital clock widget is presented.



Figure 8. Clock widget

The **Clock** widget is a LCD that displays the current time and updates itself automatically. A colon separator blinks to indicate the passing seconds.

In `clock.h`, **Clock** is defined like this:

```

#include <qlcdnumber.h>

class Clock : public QLCDNumber
{
public:
    Clock( QWidget *parent = 0, const char *name = 0 );

protected:
    void timerEvent( QTimerEvent *event );

private:
    void showTime();

    bool showingColon;
};

```

**Clock** inherits its LCD functionality from the **QLCDNumber** widget. It has a constructor typical of widget classes, with optional `parent` and `name` parameters. (Testing and debugging are easier if `name` is set.) The `timerEvent()` function is inherited from **QObject** and is called at regular intervals by the system.

In `clock.cpp`, the functions declared in `clock.h` are implemented:

```

#include <qdatetime.h>

#include "clock.h"

Clock::Clock( QWidget *parent, const char *name )
    : QLCDNumber( parent, name ), showingColon( TRUE )
{
    showTime();
    startTimer( 1000 );
}

```



```

void Clock::timerEvent( QTimerEvent * )
{
    showTime();
}

void Clock::showTime()
{
    QString time = QTime::currentTime().toString().left( 5 );
    if ( !showingColon )
        time[2] = ' ';
    display( time );
    showingColon = !showingColon;
}

```

The constructor calls `showTime()` to initialize the clock with the current time, and tells the system to call `timerEvent()` every 1000 milliseconds to refresh the LCD display.

In `showTime()`, `QLCDNumber::display()` is called with the current time. The colon is replaced by a space every other time `showTime()` is called to make the colon blink.

The `clock.h` and `clock.cpp` files completely define and implement the **Clock** custom widget. This widget can be used straight away in a simple program:

```

#include <qapplication.h>

#include "clock.h"

int main( int argc, char **argv )
{
    QApplication app( argc, argv );
    Clock *clock = new Clock;
    app.setMainWidget( clock );
    clock->show();
    return app.exec();
}

```

This example program contains a single widget (the **Clock**) and no child widgets. Complex widgets are built by combining widgets in layouts.

Developers can also write custom widgets from scratch. For example, to create an analog clock, it would be necessary to draw the clock's face and hands in code rather than relying on the functionality implemented in a base class. This approach is covered in 2D Graphics [p. 22].

## On-line References

<http://doc.trolltech.com/3.0/qwidget.html>

## 3. Signals and Slots

*Signals and slots provide inter-object communication. They are easy to understand and use, and are fully supported by Qt Designer.*

GUI applications respond to user actions. For example, when a user clicks a menu item or toolbar button, the application executes some code. More generally, we want objects of any kind to be able to communicate with each other. The programmer must relate events to the relevant code. Older toolkits use mechanisms that are not type safe (i.e. are crash prone), are inflexible, and are not object oriented.

Trolltech has invented a solution called ‘signals and slots’. Signals and slots is a powerful inter-object communication mechanism that can be used to completely replace the crude callbacks and message maps used by legacy toolkits. Signals and slots are type safe, flexible, fully object oriented and implemented in C++.

To associate some code with a button using the old callback mechanism, it is necessary to pass a pointer to a function to the button. When the button is clicked, the function is then called. Old toolkits do not ensure that arguments of the right type are given to the function when it is called, which makes crashes more likely. Another problem with the callback approach is that it tightly binds the GUI element to the functionality, making it difficult to develop classes independently.

Qt’s signals and slots mechanism is different. Qt widgets emit signals when events occur. For example, a button will emit a ‘clicked’ signal when it is clicked. The programmer can choose to connect to a signal by creating a function (called a slot) and calling the `connect()` function to relate the signal to the slot. Qt’s signals and slots mechanism does not require classes to have knowledge of each other, which makes it much easier to develop highly reusable classes. Signals and slots are type safe, with type errors being reported by warnings rather than by crashes.

For example, if a Quit button’s `clicked()` signal is connected to the application’s `quit()` slot, a user’s click on Quit makes the application terminate. In code, this is written as

```
connect( button, SIGNAL(clicked()), qApp, SLOT(quit()) );
```

Connections can be added or removed at any time during the execution of a Qt application.

The signals and slots implementation smoothly extends C++’s syntax and takes full advantage of C++’s object-orientated features. Signals and slots are type safe, can be overloaded or reimplemented and may appear in the public, protected or private sections of a class.

### 3.1. A Signals and Slots Example

To benefit from signals and slots, a class must inherit from **QObject** or one of its subclasses and include the `Q_OBJECT` macro in the class’s definition. Signals are declared in the `signals` section of the class, while slots are declared in the `public slots`, `protected slots` or `private slots` sections.

Here’s an example **QObject** subclass:

```
class BankAccount : public QObject
{
    Q_OBJECT
public:
    BankAccount() { curBalance = 0; }
    int balance() const { return curBalance; }
public slots:
    void setBalance( int newBalance );

signals:
    void balanceChanged( int newBalance );

private:
    int curBalance;
};
```

In the style of most C++ classes, the class **BankAccount** has a constructor, a get function `balance()`, and a set function `setBalance()`.

The class also has a signal `balanceChanged()`, which announces that the balance in the account has changed. Signals are not implemented; when a signal is emitted, the slots it is connected to are executed.

The set function is declared in the `public slots` section, so it is a slot. Slots are standard member functions with an implementation that can be called like any other function, and which can also be connected to signals.

Here's the implementation of the slot `setBalance()`:

```
void BankAccount::setBalance( int newBalance )
{
    if ( newBalance != curBalance ) {
        curBalance = newBalance;
        emit balanceChanged( curBalance );
    }
}
```

The statement

```
emit balanceChanged( curBalance );
```

causes the `balanceChanged()` signal to be emitted with the new current balance as its argument. The keyword `emit`, like `signals` and `slots`, is provided by Qt and is transformed into standard C++ by the C++ pre-processor.

Here's an example of how to connect two **BankAccounts**:

```
BankAccount x, y;
connect( &x, SIGNAL(balanceChanged(int)),
        &y, SLOT(setBalance(int)) );
x.setBalance( 2450 );
```

When the balance in `x` is set to 2450, the `balanceChanged()` signal is emitted. The signal is received by `y`'s `setBalance()` slot, which sets `y`'s balance to 2450.

One object's signal can be connected to many different slots, and many signals can be connected to one slot in a particular object. Connections are made between signals and slots whose parameters have the same types. A slot can have fewer parameters than the signal and ignore the extra parameters.

## 3.2. Meta Object Compiler

The signals and slots mechanism is implemented in pure standard C++. The implementation uses the C++ pre-processor and the Meta Object Compiler (`moc`) included with the Qt toolkit.

The `moc` reads the application's header files and generates the necessary code to support the signals and slots mechanism. Developers never have to edit or even look at the generated code. The `qmake` build tool generates the correct Makefiles for the target platform.

In addition to handling signals and slots, `moc` supports Qt's translation mechanism, its property system and extended run-time type information. The meta object compiler also makes the cross-platform introspection of C++ programs possible.

### On-line References

<http://doc.trolltech.com/3.0/object.html>

<http://doc.trolltech.com/3.0/signalsandslots.html>

<http://doc.trolltech.com/3.0/moc.html>

## 4. GUI Applications

*Building modern GUI applications with Qt is fast and simple, and can be achieved by hand coding or by using Qt Designer, Qt's visual design tool.*

Qt provides all the classes and functions necessary to create modern GUI applications. Qt can be used to create both 'main window' style applications with a menu bar, toolbars and status bar surrounding a central area, and dialog style applications that use buttons and possibly tabs to present options and information. Qt supports both SDI (single document interface) and MDI (multiple document interface). Qt also supports drag and drop and the clipboard.

Tool bars can be moved around within the toolbar area (called the 'dock area'), dragged to other dock areas, or floated as tool palettes. This functionality is built in and requires no additional code, although programmers can apply constraints to toolbar behavior if they wish.

Qt simplifies programming. For example, if a menu option, a toolbar button and a keyboard accelerator all perform the same action, the action need only be coded once.

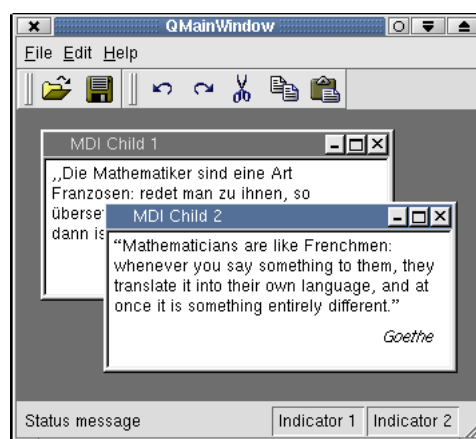
Qt also provides message boxes and a full set of standard dialogs to make it easy for applications to ask the user questions, and to get the user to choose files, folders, fonts and colors. In practice, a one-line statement using one of Qt's static convenience functions is all that is necessary to present a message box or a standard dialog.

Qt can platform-independently store application settings, such as user preferences, most recently used files, window and toolbar positions and sizes, etc.

### 4.1. Main Window Classes

#### 4.1.1. The Main Window

The **QMainWindow** class lays out a set of related widgets to provide a framework for typical application main windows.



**Figure 9.** An application main window

A main window contains a set of standard widgets. The top of the main window contains a menu bar,

beneath which toolbars are laid out. The toolbars can be moved to any dock area; main windows have dock areas at the top, left, right and bottom. Toolbars can also be dragged out of a dock area and floated as independent tool palettes. The bottom of the main window, below the bottom dock area, is occupied by a status bar. The central area contains any widget for SDI applications or a **QWorkspace** for MDI applications. Tooltips and “What’s this?” help provide balloon help for the user-interface elements.

#### 4.1.2. Menus

The **QPopupMenu** widget presents menu items to the user in a vertical list. Popup menus can be standalone (e.g. a context menu), can appear in a menu bar, or can be a sub-menu of another popup menu. Menus can also have tear-off handles.

Each menu item can have an icon, a checkbox and an accelerator. Menu items usually correspond to actions (e.g. Save). Separator items are displayed as a line and are used to group related actions visually.

Here’s an example that creates a **File** menu with *New*, *Open* and *Exit* menu items:

```
QPopupMenu *fileMenu = new QPopupMenu( this );
fileMenu->insertItem( "&New", this, SLOT(newFile()), CTRL+Key_N );
fileMenu->insertItem( "&Open...", this, SLOT(open()), CTRL+Key_O );
fileMenu->insertSeparator();
fileMenu->insertItem( "E&xit", qApp, SLOT(quit()), CTRL+Key_Q );
```

When a menu item is chosen, the corresponding slot is executed.

The **QMenuBar** class implements a menu bar. It automatically sets its geometry to the top of its parent widget (typically a **QMainWindow**). It splits its contents across multiple lines if the parent window is not wide enough. Qt’s built-in layout managers automatically take the menu bar into consideration. On the Macintosh, the menu bar appears at the top of the screen as expected.

Here’s how to create a menu bar with *File*, *Edit* and *Help* menus:

```
QMenuBar *bar = new QMenuBar( this );
bar->insertItem( "&File", fileMenu );
bar->insertItem( "&Edit", editMenu );
bar->insertItem( "&Help", helpMenu );
```

Qt’s menu system is very flexible. Menu items can be enabled, disabled, added or removed dynamically. Menu items with customized appearance and behavior can be created by subclassing **QCustomMenuItem**.

#### 4.1.3. Toolbars

The **QToolButton** class implements a toolbar button with an icon, a 3D frame and an optional label. Toggle toolbar buttons turn features on and off. Other toolbar buttons execute a command. Different icons can be provided for the active, disabled and enabled modes, and for the on and off states. If only one icon is provided, Qt automatically distinguishes the state using visual cues, for example, graying out disabled buttons. Pressing a toolbar button can also be used to trigger a popup menu.

**QToolButtons** usually appear side-by-side within a **QToolBar**. An application can have any number of toolbars, and the user is free to move them around. Toolbars can contain widgets of almost any type, for example **QComboBoxes** and **QSpinBoxes**.

#### 4.1.4. Balloon Help

Modern applications use balloon help to briefly explain the purpose of user-interface elements. Qt provides two mechanisms for balloon help: tooltips and “What’s this?” help.

Tooltips are small, usually yellow, rectangles that appear automatically when the mouse pointer hovers over a widget. Tooltips are often used to explain a toolbar button, since toolbar buttons are rarely displayed with text labels. Here’s how to set the tooltip of a ‘Save’ toolbar button:

```
QToolTip::add( saveButton, "Save" );
```

It is also possible to set a longer piece of text to be displayed in the status bar when the tooltip is shown.

“What’s this?” help is similar to tooltips, except that the user must request it, for example by pressing **Shift+F1** and then clicking a widget or menu item. “What’s this?” help is typically longer than a tooltip. Here’s how to set the “What’s this?” text for a ‘Save’ toolbar button:

```
QWhatsThis::add( saveButton, "Saves the current file." );
```

The **QToolTip** and **QWhatsThis** classes provide virtual functions that can be reimplemented for more specialized behavior. For example, to display different text according to the position of the mouse within the widget.

#### 4.1.5. Status Bar

**QStatusBar** implements a status bar. Status bars display three kinds of indicators:

1. *Temporary messages.* These are displayed at the left. Temporary messages vanish after a definable amount of time, or when another message is shown.
2. *Standard indicators.* These can be almost any widget and appear on the right. Long messages may hide them temporarily.
3. *Permanent indicators.* These appear to the right of standard indicators and are never hidden.

#### 4.1.6. Actions

Applications usually provide the user with several different ways to perform a particular action. For example, most applications provide a ‘Save’ action available from the menu (**File|Save**), from the toolbar (the ‘floppy disk’ toolbar button) and as an accelerator (**Ctrl+S**). The **QAction** class encapsulates this concept. It allows programmers to define an action in one place and then add that action to a menu or toolbar. Actions that only make sense as menu options can be added to menus directly.

The following code implements a ‘Save’ menu item, a ‘Save’ toolbar button and a ‘Save’ accelerator, all with balloon help:

```
QAction *saveAct = new QAction( "Save", saveIcon, "&Save",  
                               CTRL+Key_S, this );  
connect( saveAct, SIGNAL(activated()), this, SLOT(save()) );  
saveAct->setWhatsThis( "Saves the current file." );  
saveAct->addTo( fileMenu );  
saveAct->addTo( toolbar );
```

In addition to avoiding duplication, using a **QAction** ensures that the state of menu items stays in sync with the state of toolbar buttons, and that tool tips are displayed when necessary. Disabling an action will

disable any corresponding menu items and toolbar buttons. Similarly, if the user clicks a toggle toolbar button, the corresponding menu item will be checked or unchecked accordingly.

#### 4.1.7. The Central Widget

The central area of a **QMainWindow** can contain any widget. For example, a text editor could use a **QTextEdit** as its central widget:

```
QTextEdit *editor = new QTextEdit( mainWindow );
mainWindow->setCentralWidget( editor );
```

### 4.2. Multiple Document Interface

Multiple Document Interface (MDI) is provided by the **QWorkspace** class. **QWorkspace** is typically used as the central widget of a **QMainWindow**, but it can be used like any other widget.

Child widgets of **QWorkspace** are also ordinary widgets of any type. They are rendered with a frame similar to the frame around top-level widgets. Functions such as **show()**, **hide()**, **showMaximized()** and **setCaption()** work in the same way for child MDI widgets as for ordinary top-level widgets.

**QWorkspace** provides positioning strategies such as cascade and tile. If the child widgets extend outside the MDI area, scroll bars can be set to appear automatically. If a child widget is maximized, the frame buttons (e.g. Minimize) are shown in the menu bar.

### 4.3. Dialogs

Most GUI applications use dialog boxes to interact with the user for certain operations. Qt includes ready-made dialog classes with convenience functions for the most common tasks. Many aspects of the dialogs can be configured. For example, setting the text used by the buttons and labels in a **QMessageBox**.

Screenshots of some of Qt's standard dialogs are presented below.

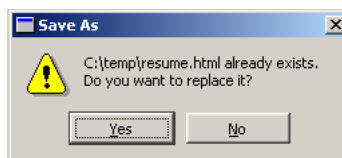


Figure 10. A **QMessageBox**

**QMessageBox** is used to provide the user with information or to present the user with simple choices (e.g. 'Yes' or 'No').

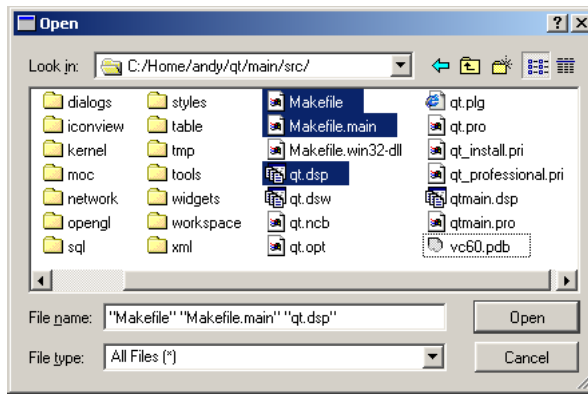


Figure 11. A **QFileDialog**

**QFileDialog** is a sophisticated file selection dialog. It can be used to select single or multiple local or remote files (e.g. using FTP), and includes functionality such as directory and file renaming and directory creation. Like most Qt dialogs, **QFileDialog** is resizable, which makes it easy to view long file names and large directories. Applications can be set to automatically use the native file dialog when compiled for Windows or Macintosh.

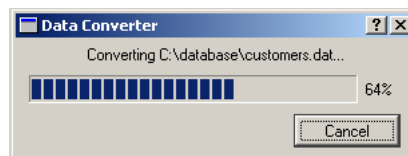


Figure 12. A **QProgressDialog**

**QProgressDialog** displays a progress bar and a ‘Cancel’ button.



Figure 13. A **QWizard**

**QWizard** provides a framework for wizard dialogs.



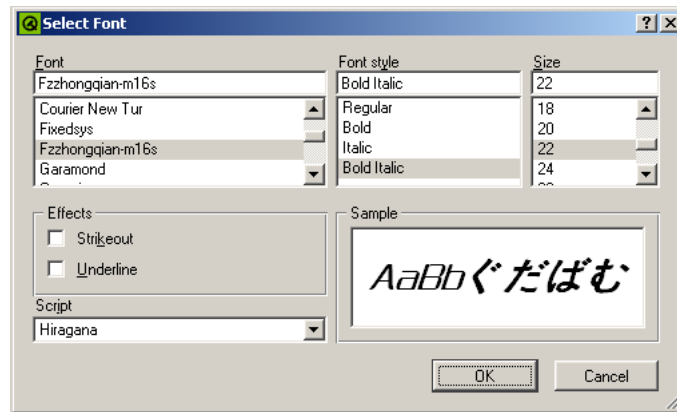


Figure 14. A QFontDialog

**QFontDialog** is used to select a font.

Qt also provides standard dialogs for color selection and printing options.

Dialogs operate in one of three ways:

1. A *modal* dialog blocks input to the other visible windows in the same application. Users must close the dialog before they can access any other window in the application.
2. A *modeless* dialog operates independently of other windows. There is little difference between a modeless **QDialog** and a **QWidget**.
3. A *semi-modal* dialog returns control to the caller immediately. These dialogs behave like modal dialogs from the user's point of view, but allow the application to continue processing. This is particularly useful for progress dialogs.

Modal dialogs are typically used like this:

```
QFileDialog dialog( workingDirectory, "", 0, 0, TRUE );
dialog.setFilter( "Text files (*.txt)" );
if ( dialog.exec() == QDialog::Accepted ) {
    do_something_with_file( dialog.selectedFile() );
}
```

Programmers can create their own dialogs by subclassing **QDialog**, which inherits **QWidget**.

#### 4.4. Dock Windows

Dock windows are windows that the user can move inside a dock area or from one dock area to another. The user can undock a dock window and make it float on top of the application or minimize it. Dock windows and areas are provided by the **QDockWindow** and **QDockArea** classes.

Qt provides one **QDockWindow** subclass, **QToolBar**. **QMainWindow** automatically provides four dock areas, one on each side of the central widget.

Developers can create custom dock windows by instantiating a **QDockWindow** object and by adding widgets to it. The widgets are laid out side-by-side if the dock area is horizontal (e.g. at the top of the main window) and above each other if the area is vertical (e.g. at the left of the main window).

Dock areas are not bound to **QMainWindow**; developers can use **QDockArea** in any custom widget. Toolbars and other dock windows can be used with any dock area.

Some applications, including *Qt Designer* [p. 18] and Qt Linguist [p. 34], use dock windows extensively. **QDockArea** provides operators to save and restore the position of dock windows, so that applications can easily restore the user's preferred positions.

## 4.5. Settings

User settings and other application settings can easily be stored on disk using the **QSettings** class. On Windows, **QSettings** makes use of the system registry; on other platforms, settings are stored in text files.

A particular setting is stored using a key. For example, the key `/SoftwareInc/Zoomer/RecentFiles` could contain a list of recently used files. Booleans, numbers, Unicode strings and lists of Unicode strings can be stored.

## 4.6. Multi-Threading

GUI applications often use multiple threads: one thread to keep the user interface responsive, and one or many other threads to perform time-consuming activities such as reading large files and performing complex calculations. Qt can be configured to support multi-threading, and provides four threading classes: **QThread**, **QMutex**, **QSemaphore** and **QWaitCondition**.

### On-line References

<http://doc.trolltech.com/3.0/threads.html>

## 5. Qt Designer

*Qt Designer* is written in *Qt* and provides visual user-interface design. *Qt* applications can be written entirely as source code, or using *Qt Designer* to speed up development.

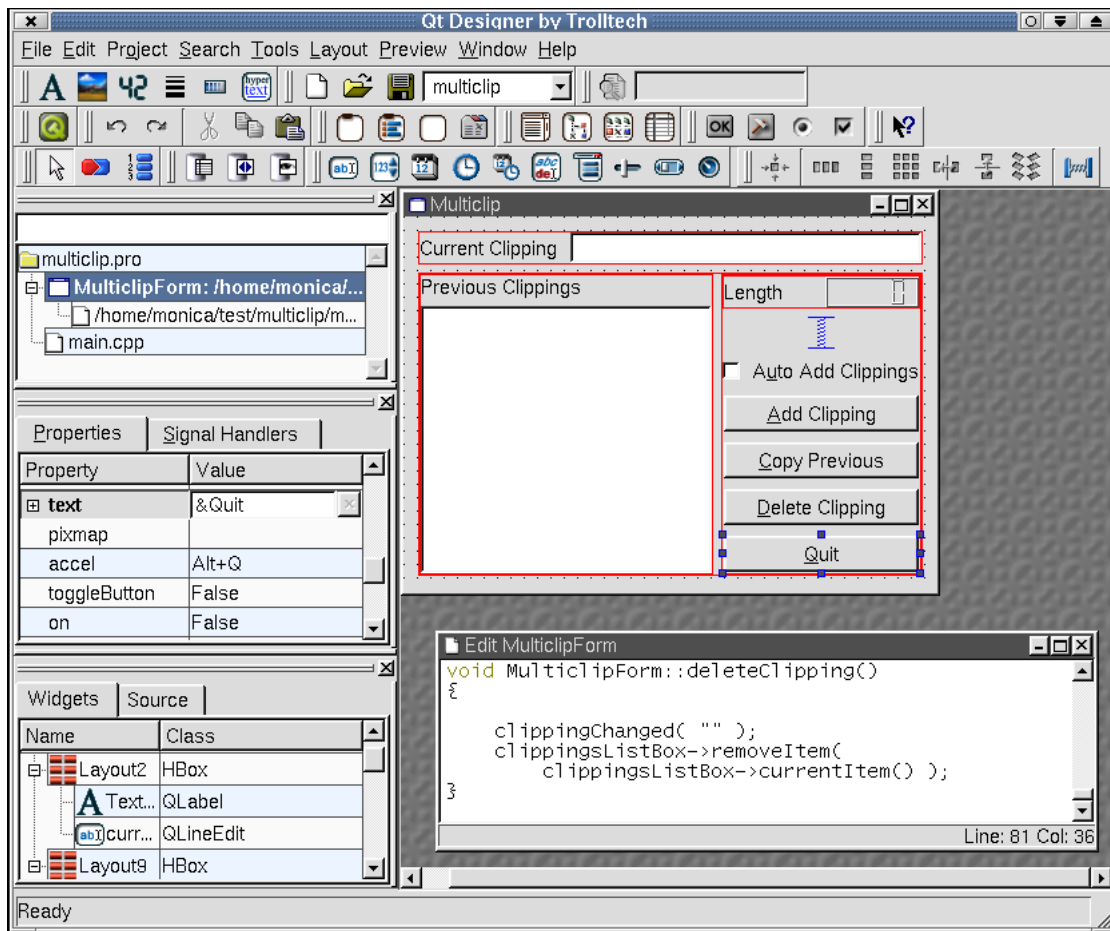


Figure 15. *Qt Designer*

Designing a form with *Qt Designer* is a simple process. Developers click a toolbar button representing the widget they want, then click on a form to place the widget. The widget's properties can then be changed using the property editor. The precise positions and sizes of the widgets does not matter. Developers select widgets and apply layouts to them. For example, some button widgets could be selected and laid out side-by-side by choosing the 'lay out horizontally' option. This approach makes design very fast, and the finished forms will scale properly to fit whatever window size the end-user prefers. See *Layouts* [p. 36] for information about Qt's automatic layouts.

*Qt Designer* eliminates the time-consuming compile, link and run cycle for user interface design. This makes it easy to correct or change designs. *Qt Designer's* preview options let developers see their forms in any style, for example, a Windows developer can preview a form in Motif style. *Qt Designer* provides live preview and editing of database data through its tight integration with Qt's database classes. See *Databases* [p. 29] for more about Qt's database support.

Developers can create both ‘dialog’ style applications and ‘main window’ style applications with menus, toolbars, balloon help, etc. Several form templates are supplied, and developers can create their own templates to ensure consistency across an application or family of applications. *Qt Designer* uses wizards to make creating toolbars, menus and database applications as fast and easy as possible. Programmers can create their own custom widgets that can easily be integrated with *Qt Designer*.

*Qt Designer* supports a project-based approach to application development. A project is represented by a `.pro` file, which `qmake` can use to generate Makefiles. Developers create a new project and then add forms and source files as required. Developers can completely separate the user interface from the underlying functionality by subclassing, or they can keep their source code and forms together by editing the form’s source directly in *Qt Designer*.

Icons and other images used in the application are automatically shared across all forms in a project to reduce executable size and speed up loading.

Form designs are stored in XML format in `.ui` files and converted into C++ header and source files by the `uic` (User Interface Compiler). The `qmake` build tool automatically includes build rules for `uic` in the Makefiles it generates, so developers do not need to invoke `uic` themselves.

Usually forms are compiled into the executable, but in some situations customers need to modify the appearance of an application without accessing the source code. Qt supports ‘dynamic dialogs’: `.ui` files that can be loaded at run-time and dynamically converted into fully functional forms. Companies can supply application executables along with the customer-modifiable forms in `.ui` format, and the customer can use *Qt Designer* to customize the appearance of the application’s forms. Loading a dynamic dialog is trivial:

```
QDialog *creditForm = (QDialog *)
                    QWidgetFactory::create( "creditform.ui" );
```

## 5.1. Qt Assistant

*Qt Designer*’s on-line help is provided by the Qt Assistant application. Qt Assistant displays Qt’s entire documentation set, and works in a similar way to a web browser. Unlike web browsers, Qt Assistant applies an intelligent indexing algorithm to provide fast searching for words and phrases that occur in the documentation.

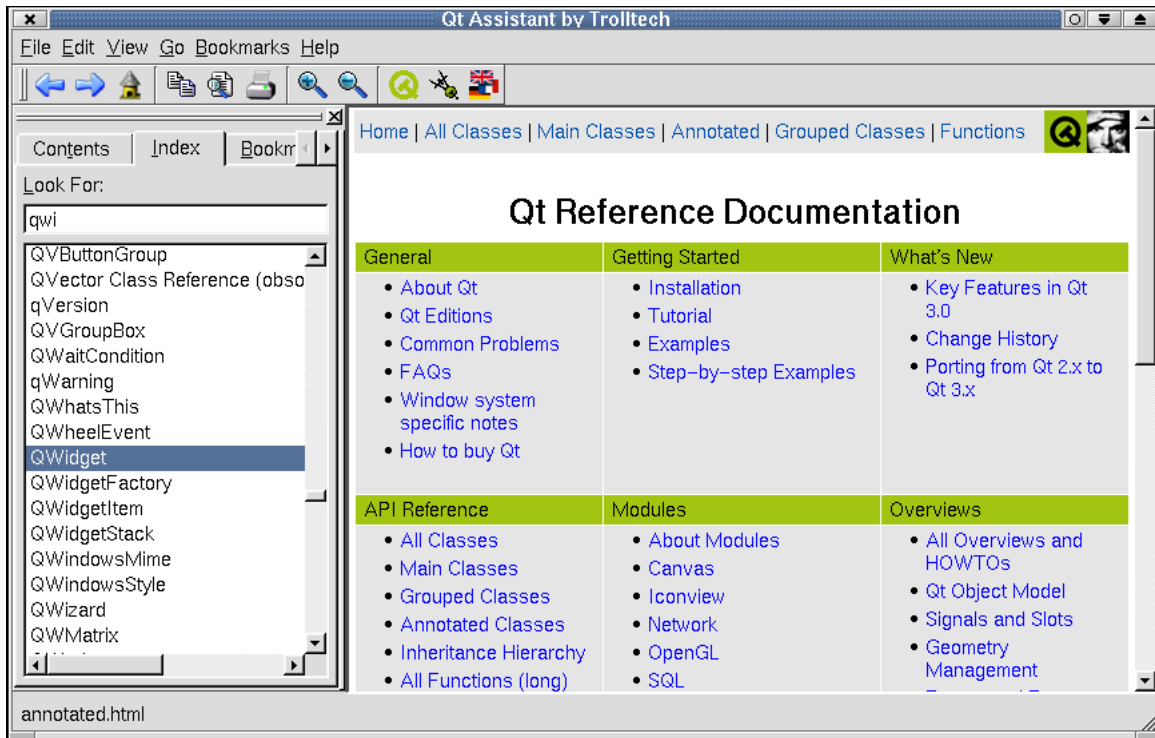


Figure 16. Qt Assistant

Qt's reference documentation consists of around 1,500 HTML pages (over 2,500 US letter sized pages), which document Qt's classes and tools, and which include overviews and introductions to various aspects of Qt programming.

Qt Assistant is a Qt application that renders Qt's HTML reference documentation using **QTextEdit**. The **QTextEdit** class supports a subset of HTML 3.2, and can also use custom tags that are created with the **QStyleSheet** class.

## 5.2. GUI Application Example

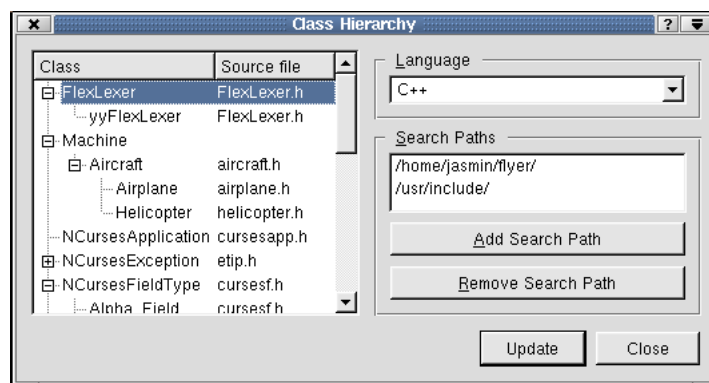


Figure 17. Class hierarchy application

The class hierarchy application is a classic 'dialog' style application where the user chooses some options,

in this case paths, and then carries out some processing based on those options.

The complete code for the application is presented below. The `main.cpp` file was produced by a *Qt Designer* wizard. The form was designed in *Qt Designer* and stored in a `.ui` file. The `.ui` file is converted into C++ by `uic`, leaving the developer free to focus on the application's functionality.

The `addSearchPath()`, `removeSearchPath()` and `updateHierarchy()` functions are all slots. They have been visually connected to the appropriate buttons using *Qt Designer*.

```
void ClassHierarchy::addSearchPath()
{
    QString path = QFileDialog::getExistingDirectory(
        QDir::homeDirPath(), this, 0, "Select a Directory" );
    if ( !path.isEmpty() &&
        searchPathBox->findItem(path, ExactMatch) == 0 )
        searchPathBox->insertItem( path );
}

void ClassHierarchy::removeSearchPath()
{
    searchPathBox->removeItem( searchPathBox->currentItem() );
}

void ClassHierarchy::updateHierarchy()
{
    QString fileNameFilter;
    QRegExp classDef;

    if ( language->currentText() == "C++" ) {
        fileNameFilter = "*.h";
        classDef.setPattern(
            "\\bclass\\s+([A-Z_a-z0-9]+)\\s*"
            "(?:\\{|:\\s*public\\s+([A-Z_a-z0-9]+))" );
    } else if ( language->currentText() == "Java" ) {
        fileNameFilter = "*.java";
        classDef.setPattern(
            "\\bclass\\s+([A-Z_a-z0-9]+)\\s+extends\\s*"
            "([A-Z_a-z0-9]+)" );
    }

    dict.clear();
    listView->clear();

    for ( int i = 0; i < searchPathBox->count(); i++ ) {
        QDir dir = searchPathBox->text( i );
        QStringList names = dir.entryList( fileNameFilter );

        for ( int j = 0; j < names.count(); j++ ) {
            QFile file( dir.filePath(names[j]) );
            if ( file.open(IO_ReadOnly) ) {
                QString content = file.readAll();
                int k = 0;
                while ( (k = classDef.search(content, k)) != -1 ) {
                    processClassDef( classDef.cap(1), classDef.cap(2),
                                    names[j] );
                    k++;
                }
            }
        }
    }
}

void ClassHierarchy::processClassDef( const QString& derived,
```

```

        const QString& base, const QString& sourceFile )
    {
        QListViewItem *derivedItem = insertClass( derived, sourceFile );

        if ( !base.isEmpty() ) {
            QListViewItem *baseItem = insertClass( base, "" );
            if ( derivedItem->parent() == 0 ) {
                listView->takeItem( derivedItem );
                baseItem->insertItem( derivedItem );
                derivedItem->setText( 1, sourceFile );
            }
        }
    }

QListViewItem *ClassHierarchy::insertClass( const QString& name,
                                           const QString& sourceFile )
{
    if ( dict[name] == 0 ) {
        QListViewItem *item = new QListViewItem( listView, name,
                                                sourceFile );

        item->setOpen( TRUE );
        dict.insert( name, item );
    }
    return dict[name];
}

```

## On-line References

<http://doc.trolltech.com/3.0/designer-manual.html>

## 6. 2D and 3D Graphics

*Qt provides excellent support for 2D and 3D graphics. Qt's 2D graphics classes support bitmapped and vector graphics. Animation and collision detection are also supported. Qt can load and save a wide and extensible range of image formats. Qt can draw Unicode rich text, rotated and sheared as required. Qt is the de-facto standard GUI toolkit for platform-independent OpenGL programming.*

### 6.1. 2D Graphics

#### 6.1.1. Images

The **QImage** class supports the input, output and manipulation of images in several formats, including BMP, GIF<sup>1</sup>, JPEG, MNG, PNG, PNM, XBM and XPM.

Many of Qt's built-in widgets can display images, for example, buttons, labels, menu items, etc. Here's how to display an icon on a push button:

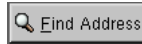
```

QPushButton *button = new QPushButton( "&Find Address", parent );
button->setIconSet( QIconSet(QImage("find.bmp")) );

```

---

<sup>1</sup>If you are in a country that recognizes software patents and where Unisys holds a patent on LZW decompression, Unisys may require you to license the technology to use GIF.



**Figure 18.** An icon on a button

**QImage** supports images with color depths of 1, 8 and 32 bits. Programmers can manipulate the pixel and palette data, apply transformations (e.g. rotations, shears, etc.) and reduce the color depth with dithering if desired. Applications can store an ‘alpha channel’ in a **QImage** along with the color data for their own purposes (e.g. transparency and alpha-blending).

The **QMovie** class can be used to display animated images.

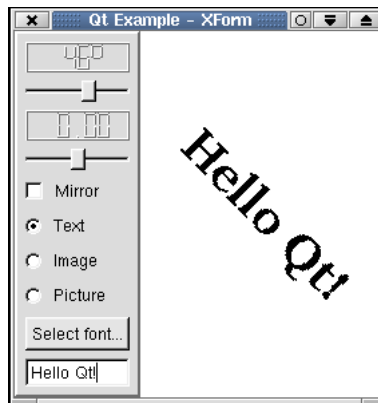
### 6.1.2. Painting

The **QPainter** provides a platform-independent API for painting widgets. It provides primitives as well as advanced functionality such as transformations and clipping. All Qt’s built-in widgets paint themselves using **QPainter**. Programmers invariably use **QPainter** when implementing their own custom widgets.

**QPainter** provides standard functions to draw points, lines, polygons, ellipses, arcs, Bezier curves, etc. The following command draws a 120 x 60 rectangle whose top-left point is at (25, 15), with a 2-pixel wide dashed red outline:

```
painter.setPen( QPen(red, 2, DashLine) );  
painter.drawRect( 25, 15, 120, 60 );
```

By default, the top-left corner of a widget is located at coordinates (0, 0), and the bottom-right corner is located at (`width()` - 1, `height()` - 1). The coordinate system of a **QPainter** object can be translated, scaled, rotated and sheared. The objects to be drawn can be clipped according to a ‘window’, and positioned on the widget using a ‘viewport’.



**Figure 19.** Qt’s `xform` example showing rotated text

The code below draws a bar-graph custom widget. It uses a **QPainter** in the reimplementation of `paintEvent()`, with the default coordinate system.



```

void BarGraph::paintEvent( QPaintEvent * )
{
    QPainter painter( this );

    draw_bar( &painter, 0, 39, Qt::DiagCrossPattern );
    draw_bar( &painter, 1, 31, Qt::BDiagPattern );
    draw_bar( &painter, 2, 44, Qt::FDiagPattern );
    draw_bar( &painter, 3, 68, Qt::SolidPattern );

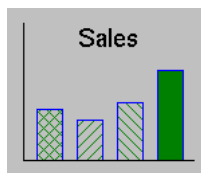
    painter.setPen( black );
    painter.drawLine( 0, 0, 0, height() - 1 );
    painter.drawLine( 0, height() - 1, width() - 1, height() - 1 );

    painter.setFont( QFont("Helvetica", 18) );
    painter.drawText( rect(), AlignHCenter | AlignTop, "Sales" );
}

void BarGraph::draw_bar( QPainter *painter, int month, int barHeight,
                        BrushStyle pattern )
{
    painter->setPen( blue );
    painter->setBrush( QBrush( darkGreen, pattern ) );
    painter->drawRect( 10 + 30 * month, height() - barHeight, 20,
                    barHeight );
}

```

The widget is drawn correctly at different sizes because the code uses the [width\(\)](#), [height\(\)](#) and [rect\(\)](#) functions. The widget produced by this code is shown below.



**Figure 20.** Custom widget

**QPainter** supports clipping using a region composed of rectangles, polygons, ellipses and bitmaps. Complex regions may be created by uniting, intersecting, subtracting and XOR'ing simple regions. Clipping can be used to reduce flicker when repainting.

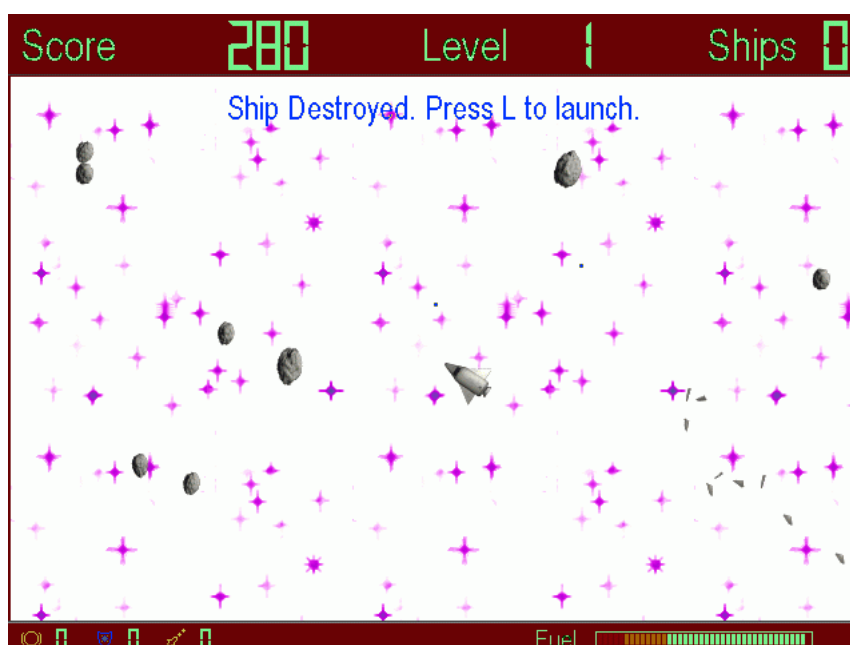
The **QColor** class stores a color specified by a RGB or HSV triple, or by a name (e.g. 'skyblue'). Qt programmers can specify any 24-bit color; Qt automatically allocates the requested color in the system's palette, or uses a similar color on color-limited displays. Commonly used colors, including red, green and blue, are predefined in Qt.

### 6.1.3. Paint Devices

**QPainter** can operate on any 'paint device'. The code required to paint on any supported device is the same, regardless of the device. Qt supports the following paint devices:

- A **QPixmap** is essentially an ‘off-screen widget’. Graphics can be painted on a **QPixmap** first, and then bit-blitted to a **QWidget** to reduce flicker. This technique is called ‘double buffering’.
- A **QPicture** is a vector image that can be scaled, rotated and sheared gracefully. The **QPicture** class stores an image as a list of paint commands rather than as pixel data. It supports the SVG (W3C’s Scalable Vector Graphics) XML format for input and output.
- A **QPrinter** represents a physical printer. On Windows, the paint commands are sent to the Windows print engine, which uses the installed printer drivers. On Unix, PostScript is output and sent to the appropriate print daemon.
- A **QWidget** is also a paint device, as shown in the earlier bar-graph example.

#### 6.1.4. Canvas



**Figure 21.** The KAsteroids game written with **QCanvas**

The **QCanvas** class provides a high-level interface to 2D graphics. It can handle a very large number of ‘canvas items’ that represent lines, rectangles, ellipses, texts, pixmaps, animated sprites, etc. Canvas items can easily be made interactive (e.g. user movable).

Canvas items are instances of **QCanvasItem** subclasses. They are more lightweight than widgets, and they can be quickly moved, hidden, and shown. **QCanvas** has efficient support for collision detection, and can list all the canvas items in a given area. **QCanvasItem** can be subclassed to provide custom item types and to extend the functionality of existing types.

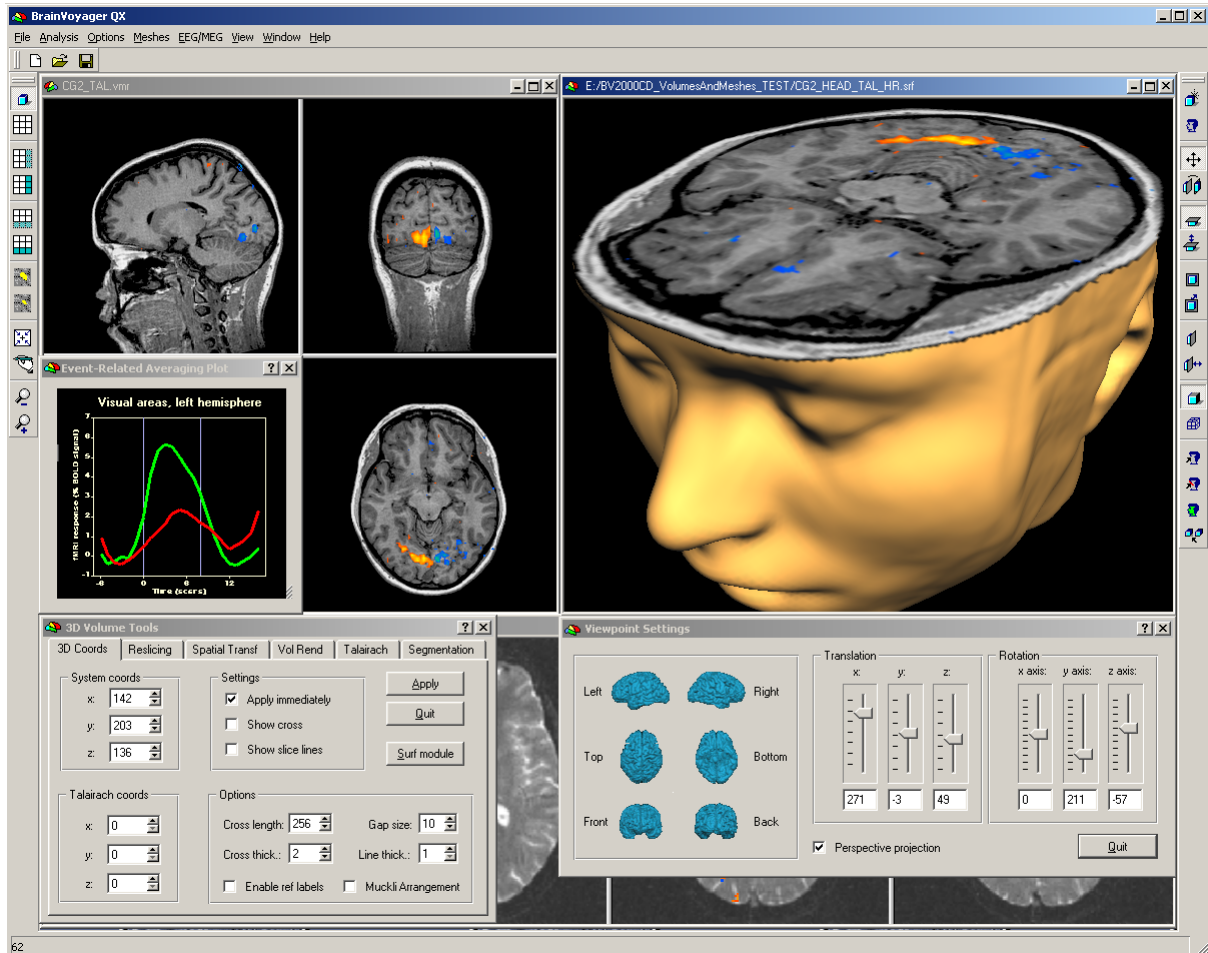
**QCanvas** objects are rendered by the **QCanvasView** class. Many **QCanvasView** objects can show the same **QCanvas**, but with different translations, scales, rotations and shears.

**QCanvas** is ideal for data visualisation. It has been used by customers for drawing road maps and for presenting network topologies. It is also suitable for fast 2D games with lots of sprites.

## 6.2. 3D Graphics

OpenGL<sup>1</sup> is a standard API for rendering 3D graphics. Qt developers can use OpenGL to draw 3D graphics in their GUI applications. This is achieved by subclassing **QGLWidget**, a **QWidget** subclass, and drawing with standard OpenGL functions rather than with **QPainter**.

Qt's OpenGL module is available on Windows, X11 and Macintosh, and uses the system's OpenGL library (including Mesa).



**Figure 22.** Brain Innovation's BrainVoyager application written in Qt and OpenGL

Qt developers can set the display format of an OpenGL rendering context: single or double buffering, depth buffer, RGBA or color index mode, alpha channel, overlays, etc. They can also set the colormap manually in color index mode.

When using Qt, developers write in pure OpenGL. Qt also provides two convenience functions, `qglClearColor()` and `qglColor()`, that accept a **QColor** argument and work in any mode.

<sup>1</sup>OpenGL is a trademark of Silicon Graphics, Inc. in the United States and other countries.

## 6.3. A 3D Example

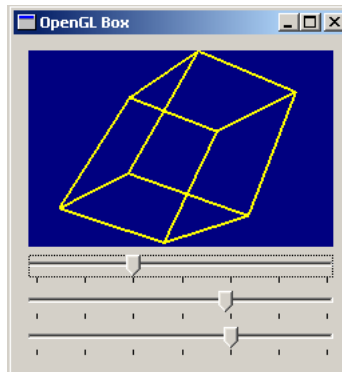


Figure 23. 3D box

The complete code for an application that draws a 3D box, with sliders to rotate the box around the X, Y and Z axes, is presented below.

In `box3d.h`, **Box3D** is defined like this:

```
#include <qgl.h>

class Box3D : public QGLWidget
{
    Q_OBJECT
public:
    Box3D( QWidget *parent = 0, const char *name = 0 );
    ~Box3D();

public slots:
    void setRotationX( int deg ) { rotX = deg; updateGL(); }
    void setRotationY( int deg ) { rotY = deg; updateGL(); }
    void setRotationZ( int deg ) { rotZ = deg; updateGL(); }

protected:
    virtual void initializeGL();
    virtual void paintGL();
    virtual void resizeGL( int w, int h );
    virtual GLuint makeObject();

private:
    GLuint object;
    GLfloat rotX, rotY, rotZ;
};
```

In `box3d.cpp`, the functions declared in `box3d.h` are implemented:

```
#include "box3d.h"

Box3D::Box3D( QWidget *parent, const char *name )
    : QGLWidget( parent, name )
{
    object = 0;
    rotX = rotY = rotZ = 0.0;
}
```

```

Box3D::~Box3D()
{
    makeCurrent();
    glDeleteLists( object, 1 );
}

void Box3D::initializeGL()
{
    qglClearColor( darkBlue );
    object = makeObject();
    glShadeModel( GL_FLAT );
}

void Box3D::paintGL()
{
    glClear( GL_COLOR_BUFFER_BIT );
    glLoadIdentity();
    glTranslatef( 0.0, 0.0, -10.0 );
    glRotatef( rotX, 1.0, 0.0, 0.0 );
    glRotatef( rotY, 0.0, 1.0, 0.0 );
    glRotatef( rotZ, 0.0, 0.0, 1.0 );
    glCallList( object );
}

void Box3D::resizeGL( int w, int h )
{
    glViewport( 0, 0, w, h );
    glMatrixMode( GL_PROJECTION );
    glLoadIdentity();
    glFrustum( -1.0, 1.0, -1.0, 1.0, 5.0, 15.0 );
    glMatrixMode( GL_MODELVIEW );
}

GLuint Box3D::makeObject()
{
    GLuint list = glGenLists( 1 );
    glNewList( list, GL_COMPILE );
    glColor( yellow );
    glLineWidth( 2.0 );

    glBegin( GL_LINE_LOOP );
    glVertex3f( +1.5, +1.0, +0.8 );
    glVertex3f( +1.5, +1.0, -0.8 );
    /* ... */
    glEnd();

    glEndList();
    return list;
}

```

In `main.cpp`, a **Box3D** instance and three sliders are created:

```

#include <qapplication.h>
#include <qslider.h>
#include <qvbox.h>

#include "box3d.h"

void create_slider( QWidget *parent, Box3D *box3d, const char *slot )
{
    QSlider *slider = new QSlider( 0, 360, 60, 0,
                                  QSlider::Horizontal, parent );
    slider->setTickmarks( QSlider::Below );
}

```

```

    QObject::connect( slider, SIGNAL(valueChanged(int)), box3d, slot );
}

int main( int argc, char **argv )
{
    QApplication::setColorSpec( QApplication::CustomColor );
    QApplication app( argc, argv );
    if ( !QGLFormat::hasOpenGL() )
        qFatal( "This system has no OpenGL support" );

    QVBox *parent = new QVBox;
    parent->setCaption( "OpenGL Box" );
    parent->setMargin( 11 );
    parent->setSpacing( 6 );
    Box3D *box3d = new Box3D( parent );
    create_slider( parent, box3d, SLOT(setRotationX(int)) );
    create_slider( parent, box3d, SLOT(setRotationY(int)) );
    create_slider( parent, box3d, SLOT(setRotationZ(int)) );

    app.setMainWidget( parent );
    parent->resize( 250, 250 );
    parent->show();
    return app.exec();
}

```

## On-line References

<http://doc.trolltech.com/3.0/coordsys.html>

<http://doc.trolltech.com/3.0/canvas.html>

<http://doc.trolltech.com/3.0/opengl.html>

## 7. Databases

*The Qt SQL module simplifies the creation of cross-platform GUI database applications. Programmers can easily execute SQL statements, use database-specific widgets, and make any widget data aware.*

The Qt SQL module provides a cross-platform interface for accessing SQL databases. Qt includes native drivers for Oracle, Microsoft SQL Server, Sybase Adaptive Server, PostgreSQL, MySQL and ODBC. The drivers work on all platforms supported by Qt and for which client libraries are available. Programs can access multiple databases using multiple drivers simultaneously.

Programmers can easily execute any SQL statements. Qt also provides a high-level C++ interface that programmers can use to generate the appropriate SQL statements automatically.

Any Qt widget, including custom widgets, can be made data aware. Qt also includes some database-specific convenience widgets, to simplify the creation of dialogs and windows that present records as forms or in tables. Data-aware widgets automatically support browsing, updating and deleting records. Most database designs require that new records have a unique key that cannot be guessed by Qt, so insertion usually needs a small amount of code to be written. The programmer can easily force the user to confirm actions, e.g. deletions.

Qt's SQL module is fully integrated into *Qt Designer*, which provides templates and wizards to make the creation of database forms as quick and easy as possible. The wizards can create forms with navigation buttons, and with update, insert and delete buttons.

Using the facilities that the Qt SQL module provides, it is straightforward to create database applications that use foreign key lookups, present master-detail relationships, and support drill-down.

## 7.1. Executing SQL Commands

The **QSqlQuery** class is used to directly execute *any* SQL statement. It is also used to navigate the result sets produced by `SELECT` statements.

In the example below, a query is executed, and the result set navigated using `QSqlQuery::next()`:

```
QSqlQuery query( "SELECT id, surname FROM staff" );
while ( query.next() ) {
    cout << "id:  "
         << query.value( 0 ).toInt()
         << " surname:  "
         << query.value( 1 ).toString() << endl;
}
```

Field values are indexed in the order they appear in the `SELECT` statement. **QSqlQuery** also provides the `first()`, `prev()`, `last()` and `seek()` navigation functions.

`INSERT`, `UPDATE` and `DELETE` are equally simple. Below is an `UPDATE` example:

```
QSqlQuery query( "UPDATE staff SET salary = salary * 1.10"
                " WHERE id > 1155 AND id < 8155" );
if ( query.isActive() ) {
    cout << "Pay rise given to "
         << query.numRowsAffected()
         << " staff" << endl;
}
```

For programmers who are not comfortable writing raw SQL, the **QSqlCursor** class provides a high-level interface for browsing and editing records in SQL tables or views without the need to write SQL statements. For example:

```
QSqlCursor cur( "staff" );
while ( cur.next() ) {
    cout << "id:  "
         << cur.value( "id" ).toInt()
         << " surname:  "
         << cur.value( "surname" ).toString() << endl;
}
```

**QSqlCursor** also supports the ordering and filtering that are achieved using the `ORDER BY` and `WHERE` clauses in SQL statements.

Calculated fields are useful both for real calculations (e.g. calculating totals) and for performing foreign key lookups (e.g. to display names rather than codes). Calculated fields can be created by subclassing **QSqlCursor**, adding additional **QSqlFields** with their calculated property set to `TRUE`, and by reimplementing `QSqlCursor::calculateField()`.

Database drivers usually supply data as strings, regardless of the actual datatype. Qt handles such data seamlessly using the **QVariant** class. Database drivers can be asked about the features they support, including query-size reporting and transactions. The `transaction()`, `commit()` and `rollback()` functions can be used if the database supports transactions.

## 7.2. Data-aware Widgets

**QDataTable** is a **QTable** that displays records from a result set using a **QSqlCursor**. **QDataTable**, like **QTable**, supports in-place editing. Programmers can force users to confirm all or selected changes (e.g. deletions) by setting **QDataTable**'s confirmation properties. The editor widget chosen for each type of data depends on the data type. For example, a **QLineEdit** is used for CHAR fields, whereas a **QSpinBox** is used for INTEGER fields. The programmer can override the defaults by creating a property map for the table, which matches fields (columns) to the editor widget type the programmer prefers.

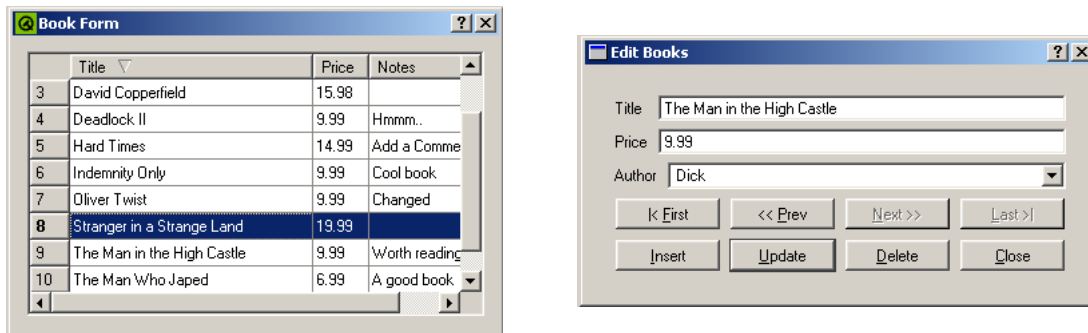


Figure 24. A **QDataTable** and a **QDataBrowser**

Records can be updated and deleted without writing any code. Insertions require some code since most database designs expect new records to be created with a unique key. This can easily be achieved by generating the key in a slot connected to the **QDataTable::beforeInsert()** signal.

**QDataTable** uses intelligent buffering to make the loading of large result sets fast, while keeping the user interface responsive. For databases that are capable of reporting query sizes, the scroll bar slider is displayed proportionally immediately.

Qt also includes **QDataBrowser** and **QDataView** to display records as forms, typically with one or perhaps just a few records shown at a time. These classes provide buttons with ready-made connections for navigating through the records. **QDataView** is used for read-only data. **QDataBrowser** is used for editing, and can provide ready-made insert, update and delete buttons.

**QDataTable** and **QDataBrowser** have both a popup context menu and keyboard shortcuts for editing records.

Programmers can manipulate data retrieved from the database before it is displayed by implementing a slot and connecting it to the **primeInsert()** and **primeUpdate()** signals. Data can also be manipulated or actions logged just before changes are written back to the database, for example, converting a foreign key's display text into its ID by implementing a slot connected to **beforeInsert()**, **beforeUpdate()** and **beforeDelete()**.

Developers can create their own forms for displaying database records. Unlike older toolkits that duplicate their widgets with data-aware versions, any Qt widget (including custom widgets) can be made data aware. All that is necessary is to include the widget in a **QSqlForm** and set up a property map to relate the relevant database field to the widget that will present and edit the field's data.

Master-detail relationships are easily set up by filtering the detail form or table's cursor by the master form or table's current record. Drill-down is also easy to achieve by associating a button, menu item or keyboard shortcut with a drill-down form that is invoked with the current record's key as a parameter.



Qt's SQL module is fully integrated with *Qt Designer*. *Qt Designer* can preview database forms and tables using live data if desired, allowing developers to browse, delete and update records. *Qt Designer* has templates and wizards to make creating database forms fast and simple.

### On-line References

<http://doc.trolltech.com/3.0/sql.html>

## 8. Internationalization

*Qt fully supports Unicode, the international standard character set. Programmers can freely mix Arabic, English, Hebrew, Japanese, Russian, and every other language supported by Unicode, in their applications. Qt also includes tools to support application translation to help companies reach international markets.*

Qt includes tools to facilitate the translation process. Programmers can easily mark user-visible text that needs translation, and a tool extracts this text from the source code. Qt Linguist is an easy-to-use GUI application that reads the extracted source texts, and provides the texts with context information ready for translation. When the translation is complete, Qt Linguist outputs a translation file for use by application programs. Qt Linguist's documentation provides the relevant information for release managers, translators and programmers.

### 8.1. Unicode

Qt uses the **QString** class to store Unicode strings, and uses it throughout the API and internally. **QString** replaces the crude `const char *` and the 16-bit **QChar** class replaces `char`. Constructors and operators are provided to automatically convert to and from 8-bit strings. Programmers can copy **QStrings** by value, since they are implicitly shared (copy on write) [p. 43], which makes them fast and memory efficient.

**QString** is more than a 16-bit character string. Functions such as **QChar::lower()** and **QChar::isPunct()** replace `tolower()` and `ispunct()` and work over the whole Unicode range. Qt's regular expression engine, provided by the **QRegExp** class, uses Unicode strings both for the regular expression pattern and the target string.

Conversion to and from different encodings and charsets is handled by **QTextCodec** subclasses. Qt uses **QTextCodec** for fonts, input methods and I/O; programmers can use it for their own purposes as well.

Qt 3.0 supports 37 different encodings, including Big5 and GBK for Chinese, EUC-JP, JIS and Shift-JIS for Japanese, KOI8-R for Russian and the ISO 8859 series; see <http://doc.trolltech.com/3.0/qtextcodec.html> for the complete list. Programmers can add their own encodings by providing a charmap or by subclassing **QTextCodec**.

### 8.2. Text Rendering

Qt provides a powerful text-rendering engine for all text that is displayed on screen, from the simplest label to the most sophisticated rich-text editor. The engine supports advanced features such as special line breaking behavior, bidirectional writing and diacritical marks. It renders most of the world's writing systems, including Arabic, Chinese, Cyrillic, English, Greek, Hebrew, Japanese, Korean, Latin and Vietnamese. Qt will automatically combine the installed fonts to render multi-language text.

### 8.3. Input Methods

Far-Eastern writing systems require many more characters than are available on a keyboard. The conversion from a sequence of key presses to actual characters is performed at the window-system level by software called ‘input methods’. Qt automatically supports the installed input methods.

### 8.4. Dates and Times

The editor classes for dates and times (**QDateEdit**, **QTimeEdit** and **QDateTimeEdit**) respect the user’s locale for separators and ordering.

### 8.5. Translating Applications

Qt provides tools and functions to help developers provide applications in their customers’ native languages.

To make a string translatable, simply wrap it in a call to `tr()` (read ‘translate’):

```
saveButton->setText( tr("Save") );
```

`tr()` attempts to replace a string literal (e.g. “Save”) with a translation if one is available; otherwise it uses the original text. English can be used as the source language and Chinese as the translated language, or vice versa. The argument to `tr()` is converted to Unicode from the application’s default encoding.

`tr()`’s general syntax is

```
Context::tr("source text", "comment")
```

The ‘context’ is the name of a **QObject** subclass. It is usually omitted, in which case the class containing the `tr()` call is used as the context. The ‘source text’ is the text to translate. The ‘comment’ is optional; along with the context, it provides additional information to human translators.

Translations are stored in **QTranslator** objects, which use disk-based `.qm` files (Qt Message files). Each `.qm` file contains the translations for a particular language. The language can be chosen at run-time, in accordance with the locale or user preferences.

Qt provides three tools for preparing `.qm` files: `lupdate`, Qt Linguist and `lrelease`.

1. `lupdate` extracts all the (context, source text, comment) triples from the source code, including *Qt Designer* `.ui` files, and generates a `.ts` file (Translation Source file). `.ts` files are in human-readable XML format.
2. Translators use Qt Linguist to provide translations for the source texts in the `.ts` files.
3. Highly compressed `.qm` files are generated by running `lrelease` on the `.ts` files.

These steps are repeated as often as necessary during the lifetime of an application. It is perfectly safe to run `lupdate` frequently, as it reuses existing translations and marks translations for obsolete source texts without eliminating them. `lupdate` also detects slight changes in source texts and automatically suggests appropriate translations. These translations are marked as unfinished so that a translator can easily check them.

Qt itself contains about 400 user-visible strings, for which Trolltech provides French and German translations.

## 8.6. Qt Linguist

Qt Linguist is a Qt application that helps translators translate Qt applications.

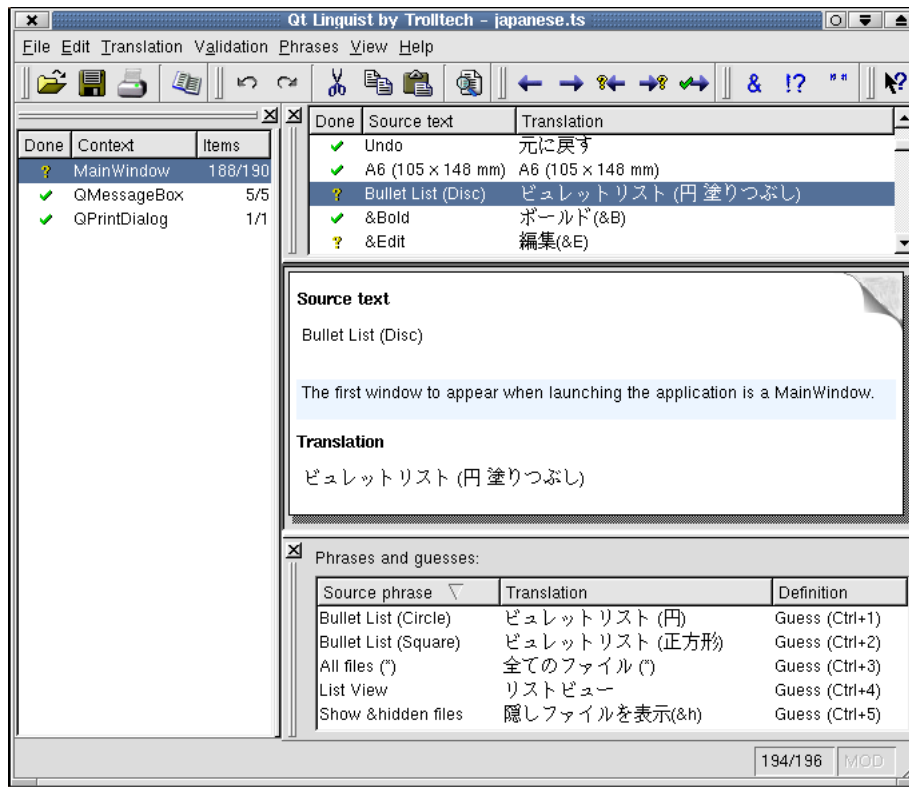


Figure 25. Qt Linguist

Translators can edit `.ts` files by hand, or more conveniently using Qt Linguist. The `.ts` file's contexts are listed in the left-hand side of the application's window. The list of source texts for the current context is displayed in the top-right area, along with translations. By selecting a source text, the translator can enter a translation; mark it done or unfinished; and proceed to the next unfinished translation. Keyboard shortcuts are provided for all the common navigation options: **Done & Next**, **Next Unfinished**, etc. The user interface's dockable windows can be reorganized to suit the translators' preferences.

Applications often use the same phrases many times in different source texts. Qt Linguist automatically displays intelligent guesses based on previously translated strings and predefined translations at the bottom of the window. Guesses often serve as a good starting point that helps translators translate similar texts consistently. Qt Linguist can optionally validate translations to ensure that accelerators and ending punctuation are translated correctly.

### On-line References

<http://doc.trolltech.com/3.0/i18n.html>  
<http://doc.trolltech.com/3.0/unicode.html>  
<http://doc.trolltech.com/3.0/scripts.html>  
<http://doc.trolltech.com/3.0/linguist-manual.html>

## 9. Styles and Themes

*Qt automatically uses the native style for look and feel. Qt applications respect user preferences for colors, fonts, sounds, etc. Qt programmers are free to use any of the supplied styles and can override any preferences. Programmers can modify existing styles or implement their own styles using Qt's powerful style engine.*

A style implements the 'look and feel' of the user interface on a particular platform. A style is a **QStyle** subclass that implements basic drawing functions such as 'draw a frame', 'draw a button', etc. Qt performs all the widget drawing itself for maximum speed and flexibility.

### 9.1. Built-in Styles

Qt provides the following built-in styles: Windows, Motif, MotifPlus, CDE, Platinum, SGI and Aqua (on the Macintosh). By default, Qt uses the appropriate style for the user's platform and desktop environment. The style can also be chosen programmatically, or with the `-style` command-line option.



**Figure 26.** Comboboxes in the different built-in styles

A style is complemented by a theme, which encapsulates the user's preferences for colors, fonts, sounds, etc. Qt automatically adapts to the computer's active theme. For example, Qt supports scroll and fade transition effects for menus and tooltips on Windows.

### 9.2. Style-aware Widgets

Qt's built-in widgets are style aware. Custom widgets and dialogs are almost always combinations of built-in widgets and layouts, and are automatically style aware. On the rare occasions that it is necessary to write a custom widget from scratch, developers can use **QStyle** to draw primitive user-interface elements rather than drawing raw rectangles directly.

### 9.3. Custom Styles

Custom styles are used to provide a distinct look to an application or family of applications. Custom styles can be defined by subclassing **QStyle**, **QCommonStyle** or any other descendent of **QCommonStyle**. It is easy to make small modifications to existing styles by reimplementing one or two virtual functions from the appropriate base class.

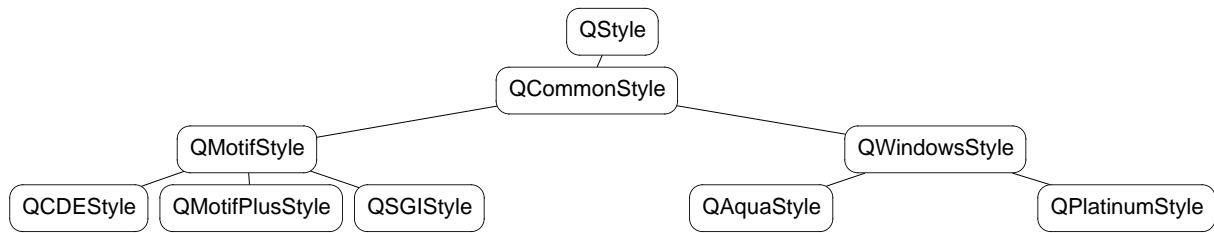


Figure 27. The full **QStyle** class hierarchy

An application's style can be set like this:

```
QApplication::setStyle( new MyCustomStyle );
```

A style can also be compiled as a plugin [p. 44]. Plugins make it possible to preview a form in a custom style in *Qt Designer* without recompiling Qt or *Qt Designer*. The style of an existing Qt application can be changed using a style plugin without recompiling the application.

### On-line References

<http://doc.trolltech.com/3.0/customstyle.html>

## 10. Layouts

*Layouts provide a powerful and flexible alternative to using fixed sizes and positions. Layouts free programmers from having to perform size and position calculations, and provide automatic scaling to suit the user's screen, language and fonts.*

Qt provides layout managers for organizing child widgets within the parent widget's area. They feature automatic positioning and resizing of child widgets, sensible minimum and default sizes for top-level widgets and automatic repositioning when the contents or the font changes. *Qt Designer* is optimized for laying out widgets using layout managers.

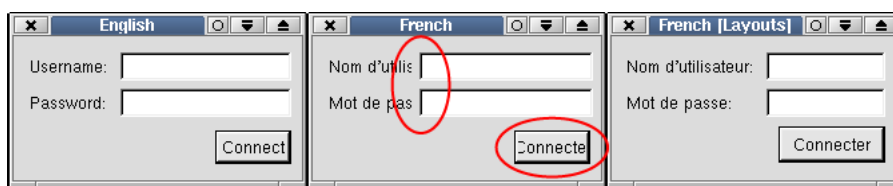


Figure 28. English, French and French with layouts

Layouts are also useful for internationalization. With fixed sizes and positions, the translation text is often truncated; with layouts, the child widgets are automatically resized.

## 10.1. Built-in Layout Managers

Qt's built-in layout managers are **QHBoxLayout**, **QVBoxLayout** and **QGridLayout**.

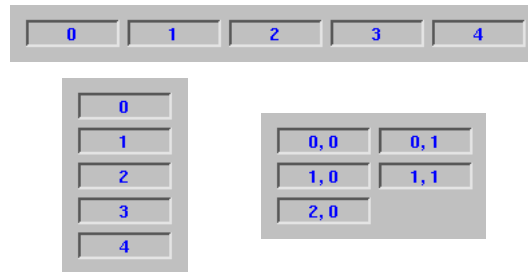


Figure 29. **QHBoxLayout**, **QVBoxLayout** and **QGridLayout**

**QHBoxLayout** organizes the managed widgets in a single horizontal row from left to right. **QVBoxLayout** organizes the managed widgets in a single vertical column, from top to bottom. **QGridLayout** organizes the managed widgets in a grid of cells; widgets may span multiple cells.

In most cases, Qt's layout managers pick optimal sizes for managed widgets so that windows resize smoothly. If the defaults are insufficient, developers can refine the layout using the following mechanisms:

1. *Setting a minimum size, a maximum size or a fixed size for some child widgets.*
2. *Adding stretch items or spacer items.* Stretch or spacer items fill empty space in a layout.
3. *Changing the size policies of the child widgets.* By calling `QWidget::setSizePolicy()`, programmers can fine tune the resize behavior of a child widget. Child widgets can be set to expand, contract, keep the same size, etc.
4. *Changing the child widgets' size hints.* `QWidget::sizeHint()` and `QWidget::minimumSizeHint()` return a widget's preferred size and preferred minimum size based on the contents. Built-in widgets provide appropriate reimplementations.
5. *Setting stretch factors.* Stretch factors allow relative growth of child widgets, e.g. two thirds of any extra space made available should be allocated to widget A and one third to widget B.

The 'spacing' between managed widgets and the 'margin' around the whole layout can also be set by the programmer. By default, *Qt Designer* sets industry-standard values based on the context.

Layouts can also run right-to-left and bottom-to-top. Right-to-left layouts are convenient for internationalized applications supporting right-to-left languages (e.g. Arabic and Hebrew).

## 10.2. Nested Layouts

Layouts can be nested to arbitrary levels. Here's an example of a dialog box, shown at two different sizes:



Figure 30. Small dialog and large dialog

The dialog uses three layouts: a **QVBoxLayout** that groups the push buttons, a **QHBoxLayout** that groups the country listbox with the push buttons and a **QVBoxLayout** that groups the “Select a country” label with the rest of the widget. A stretch item maintains the gap between the Cancel and Help buttons.

The dialog’s widgets and layouts are created with the following code:

```

QVBoxLayout *buttonBox = new QVBoxLayout( 6 );
buttonBox->addWidget( new QPushButton("OK", this) );
buttonBox->addWidget( new QPushButton("Cancel", this) );
buttonBox->addStretch( 1 );
buttonBox->addWidget( new QPushButton("Help", this) );

QList<QString> *countryList = new QList<QString>( this );
countryList->insertItem( "Canada" );
/* ... */
countryList->insertItem( "United States of America" );

QHBoxLayout *middleBox = new QHBoxLayout( 11 );
middleBox->addWidget( countryList );
middleBox->addLayout( buttonBox );

QVBoxLayout *topLevelBox = new QVBoxLayout( this, 6, 11 );
topLevelBox->addWidget( new QLabel("Select a country", this) );
topLevelBox->addLayout( middleBox );

```

Qt makes layouts so easy that programmers rarely use fixed positioning.

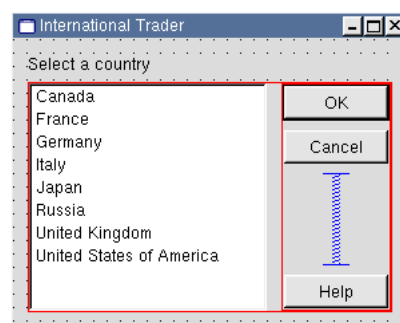


Figure 31. Laying out a form in *Qt Designer*

*Qt Designer* makes layouts even easier. With only 17 mouse clicks, you can create and lay out the widgets for the dialog shown above.

## 10.3. Custom Layouts

Developers can define custom layout managers by subclassing **QLayout**. The `customlayout` example provided with Qt presents three custom layout managers, `BorderLayout`, `CardLayout` and `SimpleFlow`, which programmers can use and modify.

Qt also includes **QSplitter**, a splitter bar that end users can manipulate. In some design situations, **QSplitter** may be preferable to a layout manager.

For complete control, it is also possible to perform layout manually in a widget by reimplementing `QWidget::resizeEvent()` and by calling `QWidget::setGeometry()` on each child widget.

### On-line References

<http://doc.trolltech.com/3.0/layout.html>

<http://doc.trolltech.com/3.0/customlayout.html>

## 11. Events

*Application objects receive system messages as Qt events. Applications can monitor, filter and respond to events at different levels of granularity.*

In Qt, an event is an object that inherits **QEvent**. Events are delivered to **QObject** objects so that they can respond to them. Programmers can monitor and filter events at the application level and at the object level.

### 11.1. Event Creation

Most events are generated by the window system and inform widgets, for example, that a key was pressed, that a mouse button was clicked or that the application window was resized. It is also possible to send simulated events to objects programmatically. There are over fifty types of event, of which the most commonly used are `MouseButtonPress`, `MouseButtonRelease`, `MouseButtonDblClick`, `Wheel`, `KeyPress`, `KeyRelease`, `Paint`, `Resize` and `Close`. Developers can add their own event types that behave like the built-in types.

It is usually insufficient merely to know that a key was pressed or that a mouse button was released. The receiver also needs to know, for example, which key was pressed, which button was released and where the mouse was located. This additional information is available from **QEvent** subclasses, such as **QMouseEvent**, **QKeyEvent**, **QPaintEvent**, **QResizeEvent** and **QCloseEvent**.

### 11.2. Event Delivery

Qt delivers events by calling the virtual function `QObject::event()`. For convenience, `QWidget::event()` forwards the most common types of event to dedicated handlers, for example, `QWidget::mousePressEvent()` and `QWidget::keyPressEvent()`. Developers can easily reimplement these handlers when writing their own widgets or when specializing existing widgets.

Some events are sent immediately, while others are queued, ready to be dispatched when control returns to the Qt kernel. Qt uses queueing to optimize certain types of events. For example, multiple paint events are compressed into a single event to minimize flicker and maximize speed.



Often an object needs to look at another object's events, e.g. to respond to them or to block them. This is achieved by having a monitoring object call `QObject::installEventFilter()` on the object that it will monitor. The monitor's `QObject::eventFilter()` virtual function will be called with each event that is destined for the monitored object before the monitored object receives the event.

It's also possible to filter all the application's events by installing a filter on `QApp`, the unique **QApplication** instance. Such filters are called before any widget-specific filters. It is even possible to reimplement `QApplication::notify()`, the event dispatcher, for complete control.

### On-line References

<http://doc.trolltech.com/3.0/eventsandfilters.html>  
<http://doc.trolltech.com/3.0/qapplication.html#notify>

## 12. Input/Output and Networking

*Qt can load and save data in plain text, XML and binary format. Qt handles local files using its own classes, and remote files using the FTP and HTTP protocols. Inter-process communication and socket-based TCP and UDP networking are also fully supported.*

### 12.1. Input/Output

Qt provides classes to perform advanced I/O on multiple platforms. The **QTextStream** class has a similar interface to the standard `<iostream>` classes, and supports the encodings provided by **QTextCodec**. The **QDataStream** class is used to serialize the basic C++ types and many Qt types in a platform-independent binary format. For example, the following code writes a Unicode string, a font and a color to the file `splash.dat`:

```
QFile file( "splash.dat" );
if ( file.open(IO_WriteOnly) ) {
    QDataStream out( &file );
    out << QString( "SplashWidgetStyle" )
        << QFont( "Times", 18, QFont::Bold )
        << QColor( "skyblue" );
}
```

The data can easily be retrieved and used, for example:

```
QString str;
QFont font;
QColor color;

QFile file( "splash.dat" );
if ( file.open(IO_ReadOnly) ) {
    QDataStream in( &file );
    in >> str >> font >> color;

    if ( str == "SplashWidgetStyle" ) {
        splashWidget->setFont( font );
        splashWidget->setColor( color );
    }
}
```

In practice, the above example would be written more concisely and easily using **QSettings**.

**QTextStream** and **QDataStream** operate on any **QIODevice** subclass. Qt includes the **QFile**, **QBuffer**, **QSocket** and **QSocketDevice** subclasses, and programmers can implement their own custom devices. **QIODevice** also provides low-level functions such as `readLine()` and `writeBlock()` that can be used independently of any stream.

Directories are read and traversed using **QDir**. **QDir** can be used to manipulate path names and access the underlying file system (e.g. create a directory or delete a file). **QFileInfo** provides more detailed information about a file, such as its size, permissions, creation time, last modification time, etc.

The following example lists the hidden files in the user's home directory along with their size, in decreasing size order:

```
QDir dir = QDir::home();
dir.setFilter( QDir::Files | QDir::Hidden );
dir.setSorting( QDir::Size | QDir::Reversed );
QStringList names = dir.entryList();

for ( int i = 0; i < names.count(); i++ ) {
    QFileInfo info( dir, names[i] );
    cout << names[i].latin1() << " "
         << info.size() << endl;
}
```

Transparent access to remote files is provided by **QUrlOperator**. In addition to local file system access, Qt supports the the FTP and HTTP protocols and can be extended to support other protocols. For example, files can be downloaded using FTP like this:

```
QUrlOperator op;
op.copy( QString("ftp://ftp.trolltech.com/qt/INSTALL"),
        QString("file:/tmp" ) );
```

URLs can easily be parsed and recomposed using **QUrl**.

Image files are usually read by creating a **QImage** with the file name as argument. Printing text and images is handled by **QPainter**. These classes are described in 2D Graphics [p. 22].

## 12.2. XML

Qt's XML module provides a SAX parser and a DOM parser, both of which read well-formed XML and are non-validating. The SAX (Simple API for XML) implementation follows the design of the SAX2 Java implementation, with adapted naming conventions. The DOM (Document Object Model) Level 2 implementation follows the W3C recommendation and includes namespace support.

Many Qt applications use XML format to store their persistent data. The SAX parser is used for reading data incrementally and is especially suitable for simple parsing requirements and for very large files. The DOM parser reads the entire file into a tree structure in memory that can be traversed at will.

## 12.3. Inter-Process Communication

The **QProcess** class is used to start external programs and to communicate with them from a Qt application in a platform-independent way. Communication is achieved by writing to the external program's standard input and potentially by reading its standard output and standard error.

**QProcess** works asynchronously, reporting the availability of data by emitting signals. Qt applications can connect to the signals to retrieve and process the data, and optionally respond by sending data back to the external program.

## 12.4. Networking

Qt provides a cross-platform interface for writing TCP/IP clients and servers.

The **QSocket** class provides an asynchronous buffered TCP connection. **QSocket** is a **QIODevice**, making it easy to use **QTextStream** and **QDataStream** on a socket.

**QSocket** is designed to work well within a GUI application. A live currency converter application illustrates this:

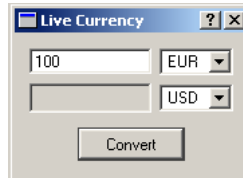


Figure 32. Live currency converter

The application uses the fictional protocol CCP (Currency Conversion Protocol) to access the latest exchange rates from a server. Only lines related to networking are presented.

```
socket = new QSocket( this );
connect( socket, SIGNAL(readyRead()),
        this, SLOT(updateTargetAmount()) );
```

The socket is created in the **Converter** constructor. Socket communication is asynchronous, and the socket emits the `readyRead()` signal when there is data available to read.

```
void Converter::convert()
{
    QString command = "CONV " + sourceAmount->text() + " " +
                      sourceCurrency->currentText() + " " +
                      targetCurrency->currentText() + "\r\n";
    socket->connectToHost( "ccp.banca-monica.nu", 123 );
    socket->writeBlock( command.latin1(), command.length() );
}
```

The `convert()` slot is called when the user clicks the Convert button. It opens the connection and sends a CONV request (e.g. CONV 100 EUR USD) to port 123 on the server `ccp.banca-monica.nu`. **QSocket** automatically uses **QDns** to resolve `ccp.banca-monica.nu` to its IP address. All these operations are non-blocking to keep the user interface responsive.

```
void Converter::updateTargetAmount()
{
    if ( socket->canReadLine() ) {
        targetAmount->setText( socket->readLine() );
        socket->close();
    }
}
```

The `updateTargetAmount()` function is called when the server replies to the CONV request. It reads the reply, updates the display and closes the connection.

Simple TCP servers can be implemented by subclassing **QServerSocket**, which works asynchronously like **QSocket**. **QServerSocket** sets up a listening socket that accepts incoming connections, and calls a virtual function to serve the client.

The **QSocketDevice** class provides a platform-independent wrapper for the native socket APIs. It provides the underlying functionality for **QSocket** and **QServerSocket**, and can be used for UDP.

### On-line References

<http://doc.trolltech.com/3.0/xml.html>

<http://doc.trolltech.com/3.0/datastreamformat.html>

## 13. Collection Classes

*Collection classes are used to store groups of items in memory. Qt provides a set of classes that are compatible with the Standard Template Library (STL), and that work regardless of whether the compiler supports STL or not.*

Applications often need to manage items in memory, for example, groups of images, widgets or custom objects. Many C++ compilers support the STL, which provides ready-made data structures for storing items. Qt provides lists, stacks, queues and dictionaries with STL-syntax. Qt's collection classes work with both STL and non-STL compilers.

Qt's rich set of portable collection classes ('containers') and associated iterators are heavily used internally, and are provided as part of the Qt API. Qt's containers are optimized for speed and memory efficiency using two techniques, 'private classes' and 'implicit sharing'. Programmers can also use STL containers on the platforms that support them, at the cost of losing Qt's optimizations.

Template classes usually increase the size of executables dramatically, because the compiler generates essentially the same code for each specialized type. Qt's template collection classes reduce code bloat because they are a thin layer over non-template private classes.

### 13.1. Value-based Collections

Qt provides five value-based collection classes: **QMap<Key,T>**, **QValueList<T>**, **QValueStack<T>**, **QValueVector<T>** and **QStringList**. They have an interface very similar to the STL containers and are fully compatible with the STL algorithms. Qt provides some STL-equivalent algorithms: **qCopy()**, **qFind()**, **qHeapSort()**, etc. On platforms with STL support, Qt provides automatic conversion operators between STL and Qt containers.

Qt's value-based collection classes are implicitly shared, also called 'copy on write'. Copies of instances of these classes share the same data in memory. The data sharing is handled automatically; if the application modifies the contents of one of the copied objects, a deep copy of the data is made so that the other objects are left unchanged. When an object is copied, only a pointer is passed and a reference count incremented, which is much faster than actually copying the data and also saves memory.

Sharing is used wherever it makes sense: in Qt's value-based collection classes, and in **QBitmap**, **QBrush**, **QCursor**, **QFont**, **QIconSet**, **QPalette**, **QPen**, **QPicture**, **QPixmap**, **QRegion**, **QRegExp**, **QString**, etc. Programmers can safely and efficiently copy objects of these classes by value, avoiding the risks related to using pointers and hand optimization. In particular, the implicitly shared **QString** class makes string processing easy and fast.

Qt also provides the low-level **QMemArray<T>** class with its subclasses **QBitArray**, **QByteArray** and **QPointArray**. These classes are very efficient for handling basic 'plain old data' types.

## 13.2. Pointer-based Collections

Qt provides many low-level, generic, pointer-based collection classes: **QDict<Key,T>**, **QPtrList<T>**, **QPtrQueue<T>**, **QPtrStack<T>**, **QPtrVector<T>** and **QCache<T>**. These classes store pointers rather than values. They are especially useful for storing pointers to **QWidgets** and **QObjects**. The pointer-based collection classes can optionally take ownership of the objects they contain and automatically delete them when the collection is destroyed.

### On-line References

<http://doc.trolltech.com/3.0/qt1.html>

<http://doc.trolltech.com/3.0/collections.html>

<http://doc.trolltech.com/3.0/shclass.html>

## 14. Plugins and Dynamic Libraries

*Qt can access functions from dynamic libraries platform-independently. Qt also supports plugins, which allow developers to create and distribute Qt codecs, database drivers, image format converters, styles and custom widgets as independent components.*

### 14.1. Dynamic Libraries

The **QLibrary** class provides cross-platform dynamic library loading, a more powerful mechanism than the more restrictive build-time linking.

Below is an example of the most basic way to dynamically load and use a library. The example attempts to obtain a pointer to the `print_str` symbol from the `mylib` library (`mylib.dll` on Windows, `mylib.so` on Unix).

```
typedef void (StrFunc)( const char *str );

QLibrary lib( "mylib" );
StrFunc *func = (StrFunc *) lib.resolve( "print_str" );
if ( func )
    func( "Hello world!" );
```

Calling a function this way is not type safe, and only symbols with C linkage are supported (due to C++ name mangling). These limitations do not apply to Qt plugins.

### 14.2. Plugins

Converting a Qt codec, database driver, image format converter, style or custom widget into a plugin is achieved by subclassing the appropriate plugin base class, implementing a few simple functions, and adding a macro.

For example, if a developer has created a **QStyle** subclass called **CopperStyle** that they want to make available as a plugin, they would create a subclass like this:

```
class CopperStylePlugin : public QStylePlugin
{
public:
    CopperStylePlugin() { }
```

```

~CopperStylePlugin() { }

QStringList keys() const {
    return QStringList() << "CopperStyle";
}

QStyle *create( const QString& key ) {
    if ( key == "CopperStyle" )
        return new CopperStyle;
    return 0;
}
};

Q_EXPORT_PLUGIN( CopperStylePlugin )

```

The new style can be set like this:

```
QApplication::setStyle( QStyleFactory::create("CopperStyle") );
```

Database drivers, codecs, custom widgets and image formats that are supplied as plugins are detected and used by the application automatically.

Companies already provide Qt components in source form, as precompiled dynamic libraries or as plugins.

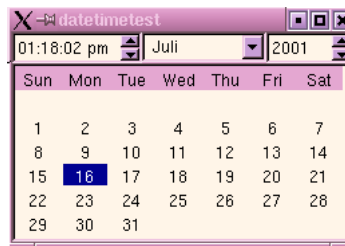


Figure 33. One of Klarälvdalens Datakonsult's many commercial components

## On-line References

<http://doc.trolltech.com/3.0/plugins-howto.html>

## 15. Qt's Architecture

*Qt's functionality is built on the low-level APIs of the platforms it supports. This makes Qt flexible and efficient.*

Qt is an 'emulating' cross-platform toolkit. All widgets are drawn by Qt, and programmers can extend or customize them by reimplementing virtual functions. Qt's widgets accurately emulate the look and feel of the supported platforms, as described in Styles and Themes [p. 35]. This technique also enables developers to derive their own custom styles to provide a distinct look for their applications.

Qt Application Source Code			
Qt API			
Qt/Windows	Qt/X11	Qt/Macintosh	Qt/Embedded
GDI	Xlib	Carbon	Embedded Linux
MS-Windows	Unix/Linux	Mac OS X	

**Figure 34.** Qt's Architecture

Qt uses the low-level APIs of the different platforms it supports. This differs from traditional 'layered' cross-platform toolkits that are thin wrappers over single-platform toolkits (e.g. MFC on Windows and Motif on X11). Layered toolkits are usually slow, since every function call to the library results in many additional calls down through the different API layers. Layered toolkits are limited by the inflexibilities of the underlying toolkits, and usually behave slightly differently on the different platforms they support, leading to obscure bugs in applications.

Qt is professionally supported, and takes advantage of the available platforms: Microsoft Windows, X11, Mac OS X and Embedded Linux. Using a single source tree, a Qt application can be converted into an executable simply by recompiling on the target platforms. Although Qt is a cross-platform toolkit, customers have found it to be easier to learn and more productive than platform-specific toolkits. Many customers use Qt for single-platform development, both on Windows and on Unix, preferring Qt's fully object-oriented approach.

### 15.1. Microsoft Windows

Qt/Windows uses the Win32 API and GDI for events and drawing primitives. Qt does not use MFC or any other toolkit. In particular, Qt does not use the inflexible 'common controls', but rather provides its own more powerful, customizable widgets. (For non-specialized uses, Qt uses the native Windows file and print dialogs.)

With Qt, the same executable works on Windows 95/98/NT4/ME/2000. Qt performs a run-time check for the Windows version, and uses the most advanced capabilities available. For example, only Windows NT4 and 2000 support rotated text natively; Qt renders rotated text on all Windows versions, and uses the native support where available. As this example demonstrates, Qt developers are insulated from changes in the Windows API.

Qt supports the Microsoft accessibility interfaces. Unlike Windows's common controls, Qt widgets can be extended without losing the accessibility information of the base widget. Custom widgets can also provide accessibility.

Qt also supports multiple screens on Microsoft Windows.

Qt/Windows customers create Qt applications using Microsoft Visual C++ and Borland C++.

### 15.2. X11

Qt/X11 uses Xlib to communicate with the X server directly. Qt does not use Xt (X Toolkit), Motif, Athena or any other toolkit.

Qt applications automatically adapt to the user's window manager or desktop environment, and have a native look and feel under Motif, SGI, CDE, GNOME and KDE. This contrasts with other Unix toolkits, which lock users into their own look and feel.

Qt provides full Unicode support [p. 32]. Qt applications automatically support both Unicode and non-Unicode fonts. Qt combines multiple X fonts to render multi-lingual text. Qt's font handling is intelligent enough to search all the installed fonts for characters unavailable in the current font.

Qt takes advantage of X extensions where they are available. Qt supports the RENDER extension for anti-aliased fonts and alpha-blending. Qt provides on-the-spot editing for X Input Methods. Qt supports multiple screens both with traditional multi-head and with Xinerama.

Qt supports the following versions of Unix: AIX, BSDI, FreeBSD, HP-UX, Irix, Linux, NetBSD, OpenBSD, Solaris, Tru64 and UnixWare. See <http://www.trolltech.com/products/platforms/> for an up-to-date list of supported compilers and operating system versions.

### 15.3. Mac OS X

New in version 3.0, Qt supports Mac OS X using the Carbon API. Qt/Mac creates a new market for customers who sell Qt applications.

Qt introduces layouts and straightforward internationalization support to the Macintosh. Qt handles files and asynchronous socket input/output in the event loop. Qt provides solid database support. Developers can create Macintosh applications using a modern object-oriented API that includes comprehensive documentation and full source code.

Macintosh developers can create applications on their favorite platform and broaden their market hugely simply by recompiling on, for example, Windows.

Qt/Mac also brings some technical benefits to Macintosh development, for example, standard OpenGL, straightforward internationalization and powerful visual design with *Qt Designer*.

### 15.4. Embedded Linux

Qt/Embedded provides its own windowing environment and writes directly to the Linux frame buffer. Qt/Embedded eliminates the need for an X server, and runs faster and with a lower memory footprint than X11-based embedded Linux devices.

Qt/Embedded uses alpha-blending for image painting and anti-aliased scalable TrueType and Type1 fonts. Qt/Embedded has flexible input handling, including hand-writing recognition, a pickboard and a virtual keyboard; it is easy to write new input methods. By selectively choosing features, the memory demands of Qt/Embedded can be tuned to between 800 KB and 3 MB in ROM.

See the Qt/Embedded whitepaper for a complete technical overview.

### 15.5. Qt's Development World

Companies and developers from around the world are joining the Qt development community every day. They have recognized that Qt's architecture lends itself to rapid application development. These developers, whether they are targeting one or many platforms, are benefiting from Qt's consistent and straightforward API, and from Qt's powerful supporting tools such as `qmake` and *Qt Designer*.

Qt has an active and helpful user community who communicate using the `qt-interest` mailing list. See <http://qt-interest.trolltech.com> to subscribe or to browse the archive.

Qt's extensive documentation is available on-line at <http://doc.trolltech.com>.

Developers can evaluate Qt, with support, for 30 days on their preferred platform.

See <http://www.trolltech.com> for details. For further information, email [info@trolltech.com](mailto:info@trolltech.com).



## Index

- About box, 14
- Accelerator, 13, 34
- Accessibility, 46
- Action, 13
- AIX, 47
- Algorithm, 43
- Alpha-blending, 47
- Alpha channel, 23, 26
- Animation, 23, 25
- Anti-aliased font, 47
- Aqua, 35
- Arabic, 32, 37
- Array, 43
- Assistant, 15, 19
- Asynchronous I/O, 41, 42
- Athena, 46
- Auto-deletion, 44
- Automatic layout, 36
- Balloon help, 13
- Bezier curve, 23
- Bidirectional writing, 32
- Big5, 32
- Binary serialization, 40
- Bitmap, 22, 25
- Bloat problem, 43
- BMP, 22
- BorderLayout, 39
- Borland C++, 46
- Box layout, 5, 37
- BSDI, 47
- Button, 5
- Cache, 44
- Calculated field, 30
- Calendar, 45
- Callback, 9
- Canvas, 25
- Caption, 14
- Carbon, 47
- CardLayout, 39
- Cascade, 14
- CDE, 35
- Central area, 12
- Central widget, 14, 16
- char, 32
- Charmap, 32
- Charset, 32
- Checkbox, 5, 6
- Child widget, 5 14, 36
- Chinese, 32
- clicked(), 9
- Clipboard, 11
- Clipping, 23, 24
- Clock, 7
- Code bloat problem, 43
- Codec, 32 44, 47
- Collection class, 43
- Collision testing, 25
- Color, 24, 35
- Color dialog, 16
- Colormap, 26
- Combobox, 5
- Comment, 33
- Commit, 30
- Common controls, 46
- Common Desktop Environment, 35
- Communication, 9
- Compiler features, 10
- Component, 9
- Configuration, 17
- connect(), 9
- Connection, 9, 13
- Container, 43
- Context, 33
- Context menu, 12, 31
- Control, 4
- Coordinate, 23
- Copy on write, 43
- Custom canvas item, 25
- Custom dock window, 16
- Custom I/O device, 41
- Custom layout, 39
- Custom style, 35
- Custom tag, 20
- Custom widget, 7 19, 23, 29 35, 44
- Cyrillic, 32
- Database, 18 29, 44
- Data Definition Language, 30
- Data Manipulation Language, 30
- Data table, 31
- Data visualisation, 25
- Date, 5, 33
- DDL, 30
- Defaults, 17
- Default widget size, 36
- Delete, 44
- Designer, 8, 17 18 32, 29 33 36 37, 38
- Diacritical mark, 32
- Dial, 5
- Dialog, 14
- Dictionary, 44
- Directory, 15, 41
- DLL, 44
- DML, 30
- Dock window, 16
- Documentation, 20, 47
- DOM, 41
- Double buffering, 25, 26
- Drag and drop, 11
- Drawing, 23 35, 39
- Drill-down, 31
- Druid, 15
- .dsp, 3
- Dynamic library, 44
- Editor, 5
- Embedded Linux, 46, 47
- Emitting a signal, 10
- Emulation, 35, 45
- Encoding, 32
- English, 32
- Error, 14
- EUC-JP, 32
- Evaluation, 47
- Event, 9 23, 39
- exec(), 5
- Fade effect, 35
- Fatal error, 14
- File dialog, 15, 46
- Fixed positioning, 38
- Flicker, 24 25, 39
- Flow layout, 39
- Font, 32, 35 36, 47
- Font dialog, 16
- Foreign key, 30, 31
- Form, 31
- Frame, 14
- Frame buffer, 47
- FreeBSD, 47
- French, 33
- FTP, 41
- Game, 25
- GBK, 32
- GCC, 47
- GDI, 46

Geometry, 5, 36  
 German, 33  
 GIF, 22  
 GL, 26  
 Graph, 25  
 Graphics, 22  
 Greek, 32  
 Grid layout, 37  
 GUI application, 11  
 Guide, 20, 47  
 Hebrew, 32, 37  
 height(), 23, 24  
 Hierarchical tree view, 6  
 Hover help, 13  
 HP-UX, 47  
 HSV, 24  
 HTML, 5, 20  
 HTTP, 41  
 Icon, 12, 22  
 Icon view, 6  
 Image, 22 25, 44  
 Implicit sharing, 32, 43  
 Inheriting, 7 9 30 32, 12 19 25  
     35, 26 39 42, 44  
 Input method, 33, 47  
 Input/output, 40  
 Input validation, 6  
 Interface emulation, 35, 45  
 Internationalization, 32, 36  
 Introspection, 10  
 iostream, 40  
 IPC, 41  
 Irix, 47  
 ISO 8859, 32  
 Iterator, 43  
 Japanese, 32, 34  
 Java, 41  
 JIS, 32  
 JPEG, 22  
 Key, 29  
 Keyboard, 33, 39  
 KOI8-R, 32  
 Korean, 32  
 Label, 5  
 Language, 32, 36  
 Latin, 32  
 Layered toolkits, 46  
 Layout, 5, 36  
 LCD, 5, 7  
 Library, 44  
 Line breaking, 32  
 Line editor, 5  
 Linguist, 17, 33  
 Linking, 44  
 Linux, 46, 47  
 List, 43, 44  
 List box, 6  
 List view, 6  
 Locale, 33  
 Localization, 32  
 Look and feel, 35 45, 46  
 Irelease, 33  
 lupdate, 33  
 Macintosh, 12, 26 35, 46  
 Magic, 10  
 Mailing list, 47  
 Main window, 11  
 Makefile, 3 10, 19  
 Manual, 20, 47  
 Manual layout, 39  
 Map, 43  
 Margin, 37  
 Master-detail, 31  
 Maximum size, 37  
 MDI, 12, 14  
 Memory array, 43  
 Memory constraints, 47  
 Menu bar, 12, 13  
 Mesa, 26  
 Message box, 14  
 Message map, 9  
 Messaging, 39  
 Meta-file, 25  
 Meta Object Compiler, 10  
 MFC, 9, 46  
 Microsoft SQL Server, 29  
 Microsoft Visual C++, 46  
 Microsoft Windows, 26 35, 46  
 Minimum size, 37  
 MNG, 22  
 moc, 10  
 Modal dialog, 16  
 Model, 23  
 Motif, 9, 35, 46  
 MotifPlus, 35  
 Mouse, 39  
 Movie, 23  
 Multi-line editor, 5  
 Multiple Document Interface,  
     12, 14  
 Multiple screens, 46, 47  
 Multi-threading, 17  
 MySQL, 29  
 Name of widget, 7  
 Native dialog, 14  
 NetBSD, 47  
 Networking, 41, 42  
 Notebook, 15  
 notify(), 40  
 Object-oriented programming,  
     9  
 OCI, 29  
 ODBC, 29  
 OpenBSD, 47  
 OpenGL, 26, 47  
 Oracle, 29  
 Overlay, 26  
 Ownership, 44  
 Painting, 23, 39  
 Palette, 23 24, 35  
 Parent widget, 5 7, 36  
 Picture, 22, 25  
 Pixmap, 25  
 Plain old data, 43  
 Platforms, 46  
 Platinum, 35  
 Plugin, 36, 44  
 PNG, 22  
 PNM, 22  
 Pointer-based collection, 44  
 Popup menu, 12, 31  
 Positioning, 36  
 PostgreSQL, 29  
 Preferences, 17, 35  
 Preferred size, 37  
 Pre-processor, 10  
 Primary key, 29  
 Print dialog, 16, 46  
 Printer, 25  
 Private class, 43  
     .pro, 19  
 Process, 41  
 Progress bar, 5, 15  
 Property, 10  
 Property box, 15  
 Push button, 5  
 QAction, 13  
 QApplication, 5, 36 40, 45  
 QAquaStyle, 35  
 QByteArray, 43  
 QPixmap, 43  
 QBrush, 43  
 QBuffer, 41

QButtonGroup, 5  
 QByteArray, 43  
 QCache, 44  
 QCanvas, 25  
 QCanvasItem, 25  
 QCanvasView, 25  
 QCDEStyle, 35  
 QChar, 32  
 QCheckBox, 5, 6  
 QCloseEvent, 39  
 QColor, 24, 26  
 QComboBox, 5, 6  
 QCommonStyle, 35  
 qCopy(), 43  
 QCursor, 43  
 QCustomMenuItem, 12  
 QDataBrowser, 31  
 QDataStream, 40  
 QDataTable, 31  
 QDataView, 31  
 QDateEdit, 33  
 QDateTimeEdit, 5, 33  
 QDial, 5  
 QDialog, 4, 16  
 QDict, 44  
 QDir, 41  
 QDns, 42  
 QDockArea, 16  
 QDockWindow, 16  
 QEvent, 39  
 QFile, 41  
 QFileDialog, 15  
 QFileInfo, 41  
 qFind(), 43  
 QFont, 43  
 QFontDialog, 16  
 QFrame, 4  
 qglClearColor(), 26  
 qglColor(), 26  
 QGLWidget, 26  
 QGridLayout, 5, 6, 37  
 QGroupBox, 5  
 QHBoxLayout, 5, 37  
 qHeapSort(), 43  
 QIconSet, 43  
 QIconView, 6  
 QImage, 22  
 QIODevice, 41, 42  
 QKeyEvent, 39  
 QLabel, 4, 5  
 QLayout, 39  
 QLCDNumber, 5, 7  
 QLibrary, 44  
 QLineEdit, 4, 5, 6  
 QListBox, 6  
 ListView, 6  
 .qm, 33  
 QMainWindow, 11  
 qmake, 3, 10, 19  
 QMap, 43  
 QMemArray, 43  
 QMenuBar, 12  
 QMessageBox, 14  
 QMotifPlusStyle, 35  
 QMotifStyle, 35  
 QMouseEvent, 39  
 QMovie, 23  
 QMutex, 17  
 QObject, 4, 9, 33, 39, 44  
 QPainter, 23  
 QPaintEvent, 39  
 QPalette, 43  
 QPen, 43  
 QPicture, 25, 43  
 QPixmap, 25, 43  
 QPlatinumStyle, 35  
 QPointArray, 43  
 QPopupMenu, 12  
 QPrinter, 25  
 QProcess, 41  
 QProgressBar, 5  
 QProgressDialog, 15  
 QPtrList, 44  
 QPtrQueue, 44  
 QPtrStack, 44  
 QPtrVector, 44  
 QPushButton, 5  
 QRadioButton, 5, 6  
 QRegExp, 6, 32, 43  
 QRegion, 43  
 QResizeEvent, 39  
 QScrollBar, 5  
 QScrollView, 6  
 QSemaphore, 17  
 QServerSocket, 42  
 QSettings, 17  
 QSGIStyle, 35  
 QSlider, 5  
 QSocket, 41, 42  
 QSocketDevice, 41, 43  
 QSpinBox, 4, 5, 6  
 QSplitter, 39  
 QSqlCursor, 30  
 QSqlField, 30  
 QSqlForm, 31  
 QSqlQuery, 30  
 QStatusBar, 13  
 QString, 32, 40, 43  
 QStringList, 43  
 QStyle, 35, 44  
 QStyleSheet, 20  
 QTabDialog, 15  
 QTable, 6  
 Qt Assistant, 19  
 Qt Designer, 8, 17, 18, 32, 29, 33, 36, 37, 38  
 QTextCodec, 32, 40  
 QTextEdit, 5, 6, 14, 20  
 QTextStream, 40  
 QThread, 17  
 QTimeEdit, 33  
 QTimer, 4  
 QTL, 43  
 Qt Linguist, 17, 33  
 QToolBar, 12, 16  
 QToolButton, 12  
 QToolTip, 13  
 QTranslator, 33  
 Qt Template Library, 43  
 Query, 30  
 Queue, 44  
 quit(), 9  
 QUrl, 41  
 QUrlOperator, 41  
 QValidator, 6  
 QValueList, 43  
 QValueStack, 43  
 QValueVector, 43  
 QVariant, 30  
 QVBoxLayout, 37  
 QWaitCondition, 17  
 QWhatsThis, 13  
 QWidget, 4, 16, 25, 44  
 QWindowsStyle, 35  
 QWizard, 15  
 QWorkspace, 12, 14  
 Radio button, 5, 6  
 Rapid application development, 47  
 rect(), 24  
 Reference counting, 43  
 Reference documentation, 20, 47

- Registry, 17
- Regular expression, 6, 32
- Relative growth, 37
- RENDER, 47
- Repainting, 39
- Repositioning, 36
- Resizing, 36, 39
- Reusability, 9
- RGB, 24
- Rich text, 5
- Right-to-left languages, 32, 37
- Rollback, 30
- Rotation, 23, 25, 46
- RTTI, 10
- Run-time type information, 10
- SAX, 41
- Scale, 23, 25
- Scroll bar, 5, 6
- Scroll effect, 35
- Scroll view, 5, 6
- SDI, 12
- SELECT, 30
- Semi-modal dialog, 16
- Separator item, 12
- Serialization, 40
- Settings, 17, 35
- SGI, 35
- Shared library, 44
- Sharing, 32, 43
- Shear, 23, 25
- Shift-JIS, 32
- Signal, 9
- SimpleFlow, 39
- Single Document Interface, 12
- Size, 36
- Size policy, 37
- Slider, 5
- Slot, 9
- Socket, 41
- Solaris, 47
- Sound, 35
- Source text, 33
- Spacer item, 37
- Spacing, 37
- Spin box, 5
- Splitter, 39
- Spreadsheet, 6
- Sprite, 25
- SQL, 29
- Stack, 43, 44
- Standard Template Library, 43
- Status bar, 12, 13
- STL, 43
- Stream, 40
- Stretch, 37
- Stretch factor, 37
- String, 32, 43
- Style, 35 44, 45
- Subclassing, 7 9 30 32, 12 19 25 35, 26 39 42, 44
- Sub-menu, 12
- Support, 47
- SVG, 25
- Sybase, 29
- System registry, 17
- System sound, 35
- Table, 6, 31
- Tab widget, 15
- TCP, 42
- TDS, 29
- Tear-off handle, 12
- Template, 43
- Text editor, 5
- Text rendering, 32
- Text translation, 33
- Theme, 35, 45
- Tile, 14
- Time, 5, 33
- Timer, 7
- Toggle button, 12
- Toolbar, 11, 12 13, 16
- Tooltip, 13, 35
- tr(), 33
- Transaction, 30
- Transformation, 23 25, 46
- Transition effect, 35
- Translation, 10, 33
- Transparency, 23
- Tree view, 6
- Tru64, 47
- .ts, 33
- Type safety, 9
- UDP, 42, 43
- .ui, 19, 33
- uic, 19
- Unicode, 17, 32 40, 47
- Unisys, 22
- Unix, 46
- UnixWare, 47
- URL, 41
- User input, 6
- User settings, 17, 35
- Validation, 6
- Value-based collection, 43
- Variant type, 30
- Vector, 43, 44
- Vietnamese, 32
- View, 30
- Viewport, 23
- Visual C++, 46
- Visualisation, 25
- W3C, 25, 41
- Warning, 14
- What's this?, 13
- Wheel mouse, 39
- Widget, 4, 25
- Widget style, 35, 45
- width(), 23, 24
- Window, 14, 23
- Windows, 26 35, 46
- Wizard, 15, 32
- Workspace, 12, 14
- World matrix, 23, 46
- Writing system, 32
- XBM, 22
- X extensions, 47
- XIM, 47
- Xinerama, 47
- Xlib, 46
- XML, 19, 25 33, 41
- XPM, 22
- Xt, 46
- X Window System, 26, 46