6

- *Basic Layouts*
- *Splitters*
- *Widget Stacks*
- *Scroll Views*
- *Dock Windows*
- *Multiple Document Interface*

Layout Management

Every widget that is placed on a form must be given an appropriate size and position. Some large widgets may also need scroll bars to give the user access to all their contents. In this chapter, we will review the different ways of laying out widgets on a form, and also see how to implement dockable windows and MDI windows.

Basic Layouts

Qt provides three basic ways of managing the layout of child widgets on a form: absolute positioning, manual layout, and layout managers. We will look at each of these approaches in turn, using the Find File dialog shown in Figure 6.1 as our example.

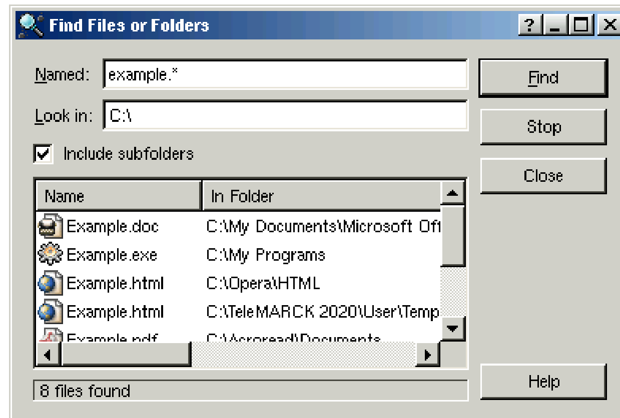

Figure 6.1. The Find File dialog

Absolute positioning is the crudest way of laying out widgets. It is achieved by assigning hard-coded sizes and positions (geometries) to the form's child widgets and a fixed size to the form. Here's what the `FindFileDialog` constructor looks like using absolute positioning:

```
FindFileDialog::FindFileDialog(QWidget *parent, const char *name)
    : QDialog(parent, name)
{
    ...
    namedLabel->setGeometry(10, 10, 50, 20);
    namedLineEdit->setGeometry(70, 10, 200, 20);
    lookInLabel->setGeometry(10, 35, 50, 20);
    lookInLineEdit->setGeometry(70, 35, 200, 20);
    subfoldersCheckBox->setGeometry(10, 60, 260, 20);
    listView->setGeometry(10, 85, 260, 100);
    messageLabel->setGeometry(10, 190, 260, 20);
    findButton->setGeometry(275, 10, 80, 25);
    stopButton->setGeometry(275, 40, 80, 25);
    closeButton->setGeometry(275, 70, 80, 25);
    helpButton->setGeometry(275, 185, 80, 25);

    setFixedSize(365, 220);
}
```

Absolute positioning has many disadvantages. The foremost problem is that the user cannot resize the window. Another problem is that some text may be truncated if the user chooses an unusually large font or if the application is translated into another language. And this approach also requires us to perform tedious position and size calculations.

An alternative to absolute positioning is manual layout. With manual layout, the widgets are still given absolute positions, but their sizes are made proportional to the size of the window rather than being entirely hard-coded. This can be achieved by reimplementing the form's `resizeEvent()` function to set its child widgets' geometries:

```
FindFileDialog::FindFileDialog(QWidget *parent, const char *name)
    : QDialog(parent, name)
{
    ...
    setMinimumSize(215, 170);
    resize(365, 220);
}

void FindFileDialog::resizeEvent(QResizeEvent *)
{
    int extraWidth = width() - minimumWidth();
    int extraHeight = height() - minimumHeight();

    namedLabel->setGeometry(10, 10, 50, 20);
    namedLineEdit->setGeometry(70, 10, 50 + extraWidth, 20);
    lookInLabel->setGeometry(10, 35, 50, 20);
    lookInLineEdit->setGeometry(70, 35, 50 + extraWidth, 20);
    subfoldersCheckBox->setGeometry(10, 60, 110 + extraWidth, 20);
```

```
    listView->setGeometry(10, 85,
                          110 + extraWidth, 50 + extraHeight);
    messageLabel->setGeometry(10, 140 + extraHeight,
                              110 + extraWidth, 20);
    findButton->setGeometry(125 + extraWidth, 10, 80, 25);
    stopButton->setGeometry(125 + extraWidth, 40, 80, 25);
    closeButton->setGeometry(125 + extraWidth, 70, 80, 25);
    helpButton->setGeometry(125 + extraWidth, 135 + extraHeight,
                            80, 25);
}
```

We set the form's minimum size to 215 × 170 in the `FindFileDialog` constructor and its initial size to 365 × 220. In the `resizeEvent()` function, we give any extra space to the widgets that we want to grow.

Just like absolute positioning, manual layout requires a lot of hard-coded constants to be calculated by the programmer. Writing code like this is tiresome, especially if the design changes. And there is still the risk of text truncation. The risk can be avoided by taking account of the child widgets' size hints, but that would complicate the code even further.

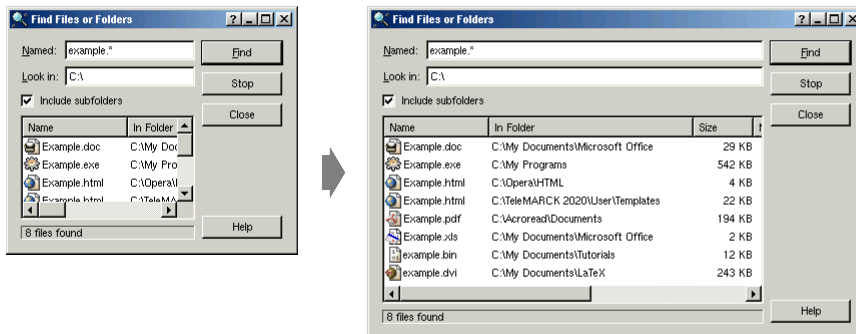

Figure 6.2. Resizing a resizable dialog

The best solution for laying out widgets on a form is to use Qt's layout managers. The layout managers provide sensible defaults for every type of widget and take into account each widget's size hint, which in turn typically depends on the widget's font, style, and contents. Layout managers also respect minimum and maximum sizes, and automatically adjust the layout in response to font changes, text changes, and window resizing.

Qt provides three layout managers: `QHBoxLayout`, `QVBoxLayout`, and `QGridLayout`. These classes inherit `QLayout`, which provides the basic framework for layouts. All three classes are fully supported by *Qt Designer* and can also be used in code. Chapter 2 presented examples of both approaches.

Here's the `FindFileDialog` code using layout managers:

```
FindFileDialog::FindFileDialog(QWidget *parent, const char *name)
    : QDialog(parent, name)
{
    ...
```

```
    QGridLayout *leftLayout = new QGridLayout;
    leftLayout->addWidget(namedLabel, 0, 0);
    leftLayout->addWidget(namedLineEdit, 0, 1);
    leftLayout->addWidget(lookInLabel, 1, 0);
    leftLayout->addWidget(lookInLineEdit, 1, 1);
    leftLayout->addMultiCellWidget(subfoldersCheckBox, 2, 2, 0, 1);
    leftLayout->addMultiCellWidget(listView, 3, 3, 0, 1);
    leftLayout->addMultiCellWidget(messageLabel, 4, 4, 0, 1);

    QVBoxLayout *rightLayout = new QVBoxLayout;
    rightLayout->addWidget(findButton);
    rightLayout->addWidget(stopButton);
    rightLayout->addWidget(closeButton);
    rightLayout->addStretch(1);
    rightLayout->addWidget(helpButton);

    QHBoxLayout *mainLayout = new QHBoxLayout(this);
    mainLayout->setMargin(11);
    mainLayout->setSpacing(6);
    mainLayout->addLayout(leftLayout);
    mainLayout->addLayout(rightLayout);
}
```

The layout is handled by one `QHBoxLayout`, one `QGridLayout`, and one `QVBoxLayout`. The `QGridLayout` on the left and the `QVBoxLayout` on the right are placed side by side by the outer `QHBoxLayout`. The margin around the dialog is 11 pixels and the spacing between the child widgets is 6 pixels.

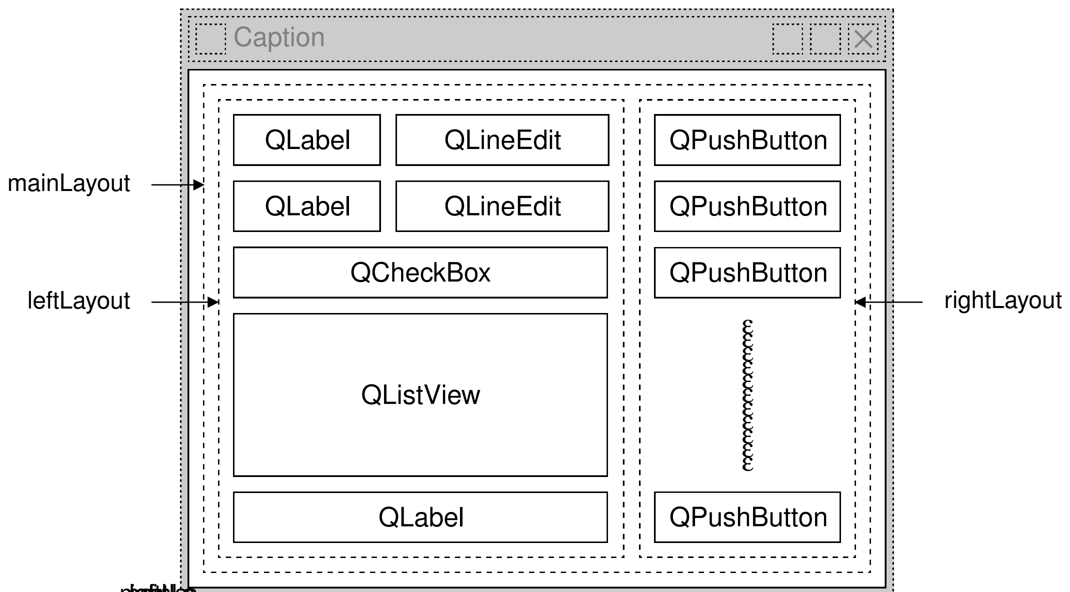

Figure 6.3. The Find File dialog's layout

`QGridLayout` works on a two-dimensional grid of cells. The `QLabel` at the top-left corner of the layout is at position (0, 0), and the corresponding `QLineEdit` is

at position (0, 1). The `QCheckBox` spans two columns; it occupies the cells in positions (2, 0) and (2, 1). The `QListView` and the `QLabel` beneath it also span two columns. The calls to `addMultiCellWidget()` have the following syntax:

```
leftLayout->addMultiCellWidget(widget, row1, row2, col1, col2);
```

where *widget* is the child widget to insert into the layout, (*row1*, *col1*) is the top-left cell occupied by the widget, and (*row2*, *col2*) is the bottom-right cell occupied by the widget.

The same dialog could be created visually in *Qt Designer* by placing the child widgets in their approximate positions, selecting those that need to be laid out together, and clicking Layout|Lay Out Horizontally, Layout|Lay Out Vertically, or Layout|Lay Out in a Grid. We used this approach in Chapter 2 for creating the Spreadsheet application's Go-to-Cell and Sort dialogs.

Using layout managers provides additional benefits to those we have discussed so far. If we add a widget to a layout or remove a widget from a layout, the layout will automatically adapt to the new situation. The same applies if we call `hide()` or `show()` on a child widget. If a child widget's size hint changes, the layout will be automatically redone, taking into account the new size hint. Also, layout managers automatically set a minimum size for the form as a whole, based on the form's child widgets' minimum sizes and size hints.

In every example presented so far, we have simply put the widgets in layouts, with spacer items to consume any excess space. Sometimes this isn't sufficient to make the layout look exactly the way we want. In such situations, we can adjust the layout by changing the size policies and size hints of the widgets being laid out.

A widget's size policy tells the layout system how it should stretch or shrink. Qt provides sensible default size policy values for all its built-in widgets, but since no single default can account for every possible layout, it is still common for developers to change the size policies for one or two widgets on a form. A size policy has both a horizontal and a vertical component. The most useful values for each component are `Fixed`, `Minimum`, `Maximum`, `Preferred`, and `Expanding`:

- `Fixed` means that the widget cannot grow or shrink. The widget always stays at the size of its size hint.
- `Minimum` means that the widget's size hint is its minimum size. The widget cannot shrink below the size hint, but it can grow to fill available space if necessary.
- `Maximum` means that the widget's size hint is its maximum size. The widget can be shrunk down to its minimum size hint.
- `Preferred` means that the widget's size hint is its preferred size, but that the widget can still shrink or grow if necessary.
- `Expanding` means that the widget can shrink or grow and that it is especially willing to grow.

Figure 6.4 summarizes the meaning of the different size policies, using a QLabel showing the text "Some Text" as an example.

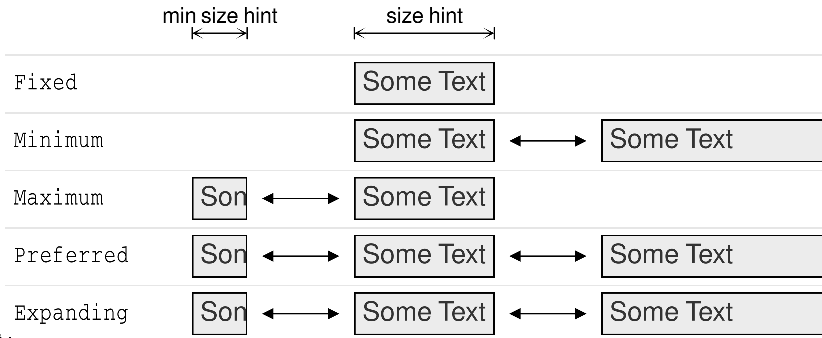

Figure 6.4. The meaning of the different size policies

When a form that contains both Preferred and Expanding widgets is resized, extra space is given to the Expanding widgets, while the Preferred widgets stay at their size hint.

There are two other size policies: MinimumExpanding and Ignored. The former was necessary in a few rare cases in older versions of Qt, but it isn't useful any more; a better approach is to use Expanding and reimplement minimumSizeHint() appropriately. The latter is similar to Expanding, except that it ignores the widget's size hint.

In addition to the size policy's horizontal and vertical components, the QSizePolicy class stores both a horizontal and a vertical stretch factor. These stretch factors can be used to indicate that different child widgets should grow at different rates when the form expands. For example, if we have a QListView above a QTextEdit and we want the QTextEdit to be twice as tall as the QListView, we can set the QTextEdit's vertical stretch factor to 2 and the QListView's vertical stretch factor to 1.

Another way of influencing a layout is to set a minimum size, a maximum size, or a fixed size on the child widgets. The layout manager will respect these constraints when laying out the widgets. And if this isn't sufficient, we can always derive from the child widget's class and reimplement sizeHint() to obtain the size hint we need.

Splitters

A splitter is a widget that contains other widgets and that separates them with splitter handles. Users can change the sizes of a splitter's child widgets by dragging the handles. Splitters can often be used as an alternative to layout managers, to give more control to the user.

Qt supports splitters with the QSplitter widget. The child widgets of a QSplitter are automatically placed side by side (or one below the other) in the

order in which they are created, with splitter bars between adjacent widgets. Here's the code for creating the window depicted in Figure 6.5:

```
#include <qapplication.h>
#include <qsplitter.h>
#include <qtextedit.h>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QSplitter splitter(Qt::Horizontal);
    splitter.setCaption(QObject::tr("Splitter"));
    app.setMainWidget(&splitter);

    QTextEdit *firstEditor = new QTextEdit(&splitter);
    QTextEdit *secondEditor = new QTextEdit(&splitter);
    QTextEdit *thirdEditor = new QTextEdit(&splitter);

    splitter.show();
    return app.exec();
}
```

The example consists of three `QTextEdit`s laid out horizontally by a `QSplitter` widget. Unlike layout managers, which simply lay out a form's child widgets, `QSplitter` inherits from `QWidget` and can be used like any other widget.

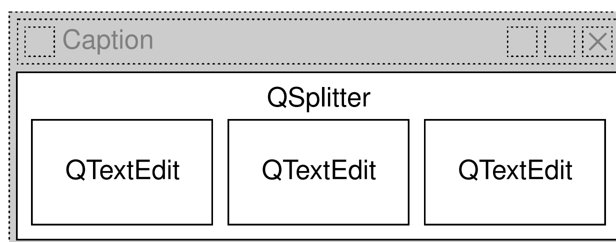

Figure 6.5. The Splitter application's widgets

A `QSplitter` can lay out its child widgets either horizontally or vertically. Complex layouts can be achieved by nesting horizontal and vertical `QSplitter`s. For example, the Mail Client application shown in Figure 6.6 consists of a horizontal `QSplitter` that contains a vertical `QSplitter` on its right side.

Here's the code in the constructor of the Mail Client application's `QMainWindow` subclass:

```
MailClient::MailClient(QWidget *parent, const char *name)
    : QMainWindow(parent, name)
{
    horizontalSplitter = new QSplitter(Horizontal, this);
    setCentralWidget(horizontalSplitter);

    foldersListView = new QListView(horizontalSplitter);
    foldersListView->addColumn(tr("Folders"));
    foldersListView->setResizeMode(QListView::AllColumns);
```

```
    verticalSplitter = new QSplitter(Vertical, horizontalSplitter);

    messagesListView = new QListView(verticalSplitter);
    messagesListView->addColumn(tr("Subject"));
    messagesListView->addColumn(tr("Sender"));
    messagesListView->addColumn(tr("Date"));
    messagesListView->setAllColumnsShowFocus(true);
    messagesListView->setShowSortIndicator(true);
    messagesListView->setResizeMode(QListView::AllColumns);

    textEdit = new QTextEdit(verticalSplitter);
    textEdit->setReadOnly(true);

    horizontalSplitter->setResizeMode(foldersListView,
                                      QSplitter::KeepSize);
    verticalSplitter->setResizeMode(messagesListView,
                                    QSplitter::KeepSize);
    ...
    readSettings();
}
```

We create the horizontal `QSplitter` first and set it to be the `QMainWindow`'s central widget. Then we create the child widgets and their child widgets.

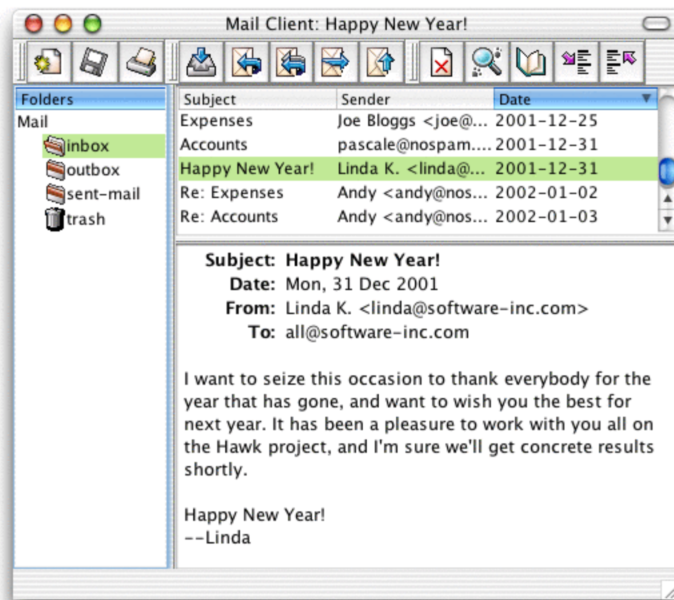

Figure 6.6. The Mail Client application on Mac OS X

When the user resizes a window, `QSplitter` normally distributes the space so that the relative sizes of the child widgets stay the same. In the Mail Client example, we don't want this behavior; instead we want the two `QListView`s to maintain their size and we want to give any extra space to the `QTextEdit`. This is achieved by the two `setResizeMode()` calls near the end.

When the application is started, `QSplitter` gives the child widgets appropriate sizes based on their initial sizes. We can move the splitter handles programmatically by calling `QSplitter::setSizes()`. The `QSplitter` class also provides a means of saving and restoring its state the next time the application is run. Here's the `writeSettings()` function that saves the Mail Client's settings:

```
void MailClient::writeSettings()
{
    QSettings settings;
    settings.setPath("software-inc.com", "MailClient");
    settings.beginGroup("/MailClient");

    QString str;
    QTextOStream out1(&str);
    out1 << *horizontalSplitter;
    settings.writeEntry("/horizontalSplitter", str);
    QTextOStream out2(&str);
    out2 << *verticalSplitter;
    settings.writeEntry("/verticalSplitter", str);

    settings.endGroup();
}
```

Here's the corresponding `readSettings()` function:

```
void MailClient::readSettings()
{
    QSettings settings;
    settings.setPath("software-inc.com", "MailClient");
    settings.beginGroup("/MailClient");

    QString str1 = settings.readEntry("/horizontalSplitter");
    QTextIStream in1(&str1);
    in1 >> *horizontalSplitter;
    QString str2 = settings.readEntry("/verticalSplitter");
    QTextIStream in2(&str2);
    in2 >> *verticalSplitter;

    settings.endGroup();
}
```

These functions rely on `QTextIStream` and `QTextOStream`, two `QTextStream` convenience subclasses.

By default, a splitter handle is shown as a rubber band while the user is dragging it, and the widgets on either side of the splitter handle are resized only when the user releases the mouse button. To make `QSplitter` resize the child widgets in real time, we would call `setOpaqueResize(true)`.

`QSplitter` is fully supported by *Qt Designer*. To put widgets into a splitter, place the child widgets approximately in their desired positions, select them, and click Layout|Lay Out Horizontally (in Splitter) or Layout|Lay Out Vertically (in Splitter).

Widget Stacks

Another useful widget for managing layouts is `QWidgetStack`. This widget contains a set of child widgets, or "pages", and shows only one at a time, hiding the others from the user. The pages are numbered from 0. If we want to make a specific child widget visible, we can call `raiseWidget()` with either a page number or a pointer to the child widget.

Figure 6.7. `QWidgetStack`

The `QWidgetStack` itself is invisible and provides no intrinsic means for the user to change page. The small arrows and the dark gray frame in Figure 6.7 are provided by *Qt Designer* to make the `QWidgetStack` easier to design with.

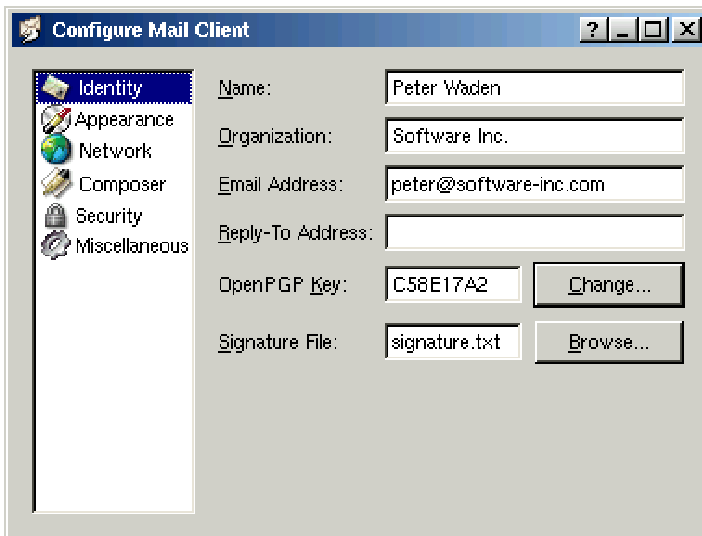

Figure 6.8. The Configure dialog

The Configure dialog shown in Figure 6.8 is an example that uses `QWidgetStack`. The dialog consists of a `QListBox` on the left and a `QWidgetStack` on the right. Each item in the `QListBox` corresponds to a different page in the `QWidgetStack`. Forms like this are very easy to create using *Qt Designer*:

1. Create a new form based on the "Dialog" or the "Widget" template.
2. Add a list box and a widget stack to the form.
3. Fill each widget stack page with child widgets and layouts.
   (To create a new page, right-click and choose Add Page; to switch pages, click the tiny left or right arrow located at the top-right of the widget stack.)
4. Lay the widgets out side by side using a horizontal layout.
5. Connect the list box's `highlighted(int)` signal to the widget stack's `raiseWidget(int)` slot.
6. Set the value of the list box's `currentItem` property to 0.

Since we have implemented page-switching using predefined signals and slots, the dialog will exhibit the correct page-switching behavior when previewed in *Qt Designer*.

Scroll Views

The `QScrollView` class provides a scrollable viewport, two scroll bars, and a "corner" widget (usually an empty `QWidget`). If we want to add scroll bars to a widget, it is much simpler to use a `QScrollView` than to instantiate our own `QScrollBar`s and implement the scrolling functionality ourselves.

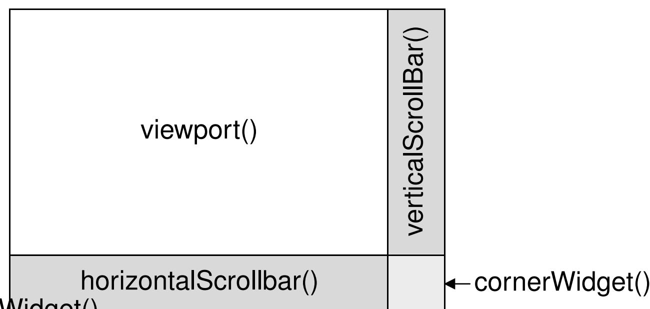

Figure 6.9. `QScrollView`'s constituent widgets

The easiest way to use `QScrollView` is to call `addChild()` with the widget we want to add scroll bars to. `QScrollView` automatically reparents the widget to make it a child of the viewport (accessible through `QScrollView::viewport()`) if it isn't already. For example, if we want scroll bars around the `IconEditor` widget we developed in Chapter 5, we can write this:

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

#include "iconeditor.h"

int main(int argc, char *argv[])
{
```

```
    QApplication app(argc, argv);

    QScrollView scrollView;
    scrollView.setCaption(QObject::tr("Icon Editor"));
    app.setMainWidget(&scrollView);

    IconEditor *iconEditor = new IconEditor;
    scrollView.addChild(iconEditor);

    scrollView.show();
    return app.exec();
}
```

By default, the scroll bars are only displayed when the viewport is smaller than the child widget. We can force the scroll bars to always be shown by writing this code:

```
scrollView.setHScrollBarMode(QScrollView::AlwaysOn);
scrollView.setVScrollBarMode(QScrollView::AlwaysOn);
```

When the child widget's size hint changes, `QScrollView` automatically adapts to the new size hint.

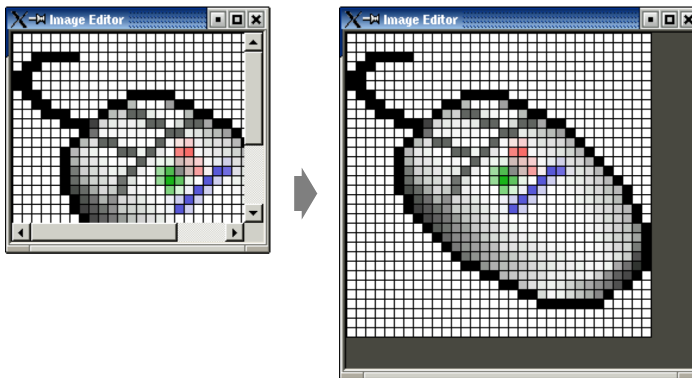

Figure 6.10. Resizing a `QScrollView`

An alternative way of using a `QScrollView` with a widget is to make the widget inherit `QScrollView` and to reimplement `drawContents()` to draw the contents. This is the approach used by Qt classes like `QIconView`, `QListBox`, `QListView`, `QTable`, and `QTextEdit`. If a widget is likely to require scroll bars, it's usually a good idea to implement it as a subclass of `QScrollView`.

To show how this works, we will implement a new version of the `IconEditor` class as a `QScrollView` subclass. We will call the new class `ImageEditor`, since its scroll bars make it capable of handling large images.

```
#ifndef IMAGEEDITOR_H
#define IMAGEEDITOR_H

#include <qimage.h>
#include <qscrollview.h>
```

```
class ImageEditor : public QScrollView
{
    Q_OBJECT
    Q_PROPERTY(QColor penColor READ penColor WRITE setPenColor)
    Q_PROPERTY(QImage image READ image WRITE setImage)
    Q_PROPERTY(int zoomFactor READ zoomFactor WRITE setZoomFactor)

public:
    ImageEditor(QWidget *parent = 0, const char *name = 0);

    void setPenColor(const QColor &newColor);
    QColor penColor() const { return curColor; }
    void setZoomFactor(int newZoom);
    int zoomFactor() const { return zoom; }
    void setImage(const QImage &newImage);
    const QImage &image() const { return curImage; }

protected:
    void contentsMousePressEvent(QMouseEvent *event);
    void contentsMouseMoveEvent(QMouseEvent *event);
    void drawContents(QPainter *painter, int x, int y,
                      int width, int height);

private:
    void drawImagePixel(QPainter *painter, int i, int j);
    void setImagePixel(const QPoint &pos, bool opaque);
    void resizeContents();

    QColor curColor;
    QImage curImage;
    int zoom;
};

#endif
```

The header file is very similar to the original (p. 100). The main difference is that we inherit from `QScrollView` instead of `QWidget`. We will run into the other differences as we review the class's implementation.

```
ImageEditor::ImageEditor(QWidget *parent, const char *name)
    : QScrollView(parent, name, WStaticContents | WNoAutoErase)
{
    curColor = black;
    zoom = 8;
    curImage.create(16, 16, 32);
    curImage.fill(qRgba(0, 0, 0, 0));
    curImage.setAlphaBuffer(true);
    resizeContents();
}
```

The constructor passes the `WStaticContents` and `WNoAutoErase` flags to the `QScrollView`. These flags are actually set on the viewport. We don't set a size policy, because `QScrollView`'s default of (`Expanding`, `Expanding`) is appropriate.

In the original version, we didn't call `updateGeometry()` in the constructor because we could depend on Qt's layout managers picking up the initial widget

size by themselves. But here we must give the `QScrollView` base class an initial size to work with, and we do this with the `resizeContents()` call.

```
void ImageEditor::resizeContents()
{
    QSize size = zoom * curImage.size();
    if (zoom >= 3)
        size += QSize(1, 1);
    QScrollView::resizeContents(size.width(), size.height());
}
```

The `resizeContents()` private function calls `QScrollView::resizeContents()` with the size of the content part of the `QScrollView`. The `QScrollView` displays scroll bars depending on the content's size in relation to the viewport's size.

We don't need to reimplement `sizeHint()`; `QScrollView`'s version uses the content's size to provide a reasonable size hint.

```
void ImageEditor::setImage(const QImage &newImage)
{
    if (newImage != curImage) {
        curImage = newImage.convertDepth(32);
        curImage.detach();
        resizeContents();
        updateContents();
    }
}
```

In many of the original `IconEditor` functions, we called `update()` to schedule a repaint and `updateGeometry()` to propagate a size hint change. In the `QScrollView` versions, these calls are replaced by `resizeContents()` to inform the `QScrollView` about a change of the content's size and `updateContents()` to force a repaint.

```
void ImageEditor::drawContents(QPainter *painter, int, int, int, int)
{
    if (zoom >= 3) {
        painter->setPen(colorGroup().foreground());
        for (int i = 0; i <= curImage.width(); ++i)
            painter->drawLine(zoom * i, 0,
                              zoom * i, zoom * curImage.height());
        for (int j = 0; j <= curImage.height(); ++j)
            painter->drawLine(0, zoom * j,
                              zoom * curImage.width(), zoom * j);
    }

    for (int i = 0; i < curImage.width(); ++i) {
        for (int j = 0; j < curImage.height(); ++j)
            drawImagePixel(painter, i, j);
    }
}
```

The `drawContents()` function is called by `QScrollView` to repaint the content's area. The `QPainter` object is already initialized to account for the scrolling

offset. We just need to perform the drawing as we normally do in a `paintEvent()`.

The second, third, fourth, and fifth parameters specify the rectangle that must be redrawn. We could use this information to only draw the rectangle that needs repainting, but for the sake of simplicity we redraw everything.

The `drawImagePixel()` function that is called near the end of `drawContents()` is essentially the same as in the original `IconEditor` class (p. 106), so it is not reproduced here.

```
void ImageEditor::contentsMousePressEvent(QMouseEvent *event)
{
    if (event->button() == LeftButton)
        setImagePixel(event->pos(), true);
    else if (event->button() == RightButton)
        setImagePixel(event->pos(), false);
}

void ImageEditor::contentsMouseMoveEvent(QMouseEvent *event)
{
    if (event->state() & LeftButton)
        setImagePixel(event->pos(), true);
    else if (event->state() & RightButton)
        setImagePixel(event->pos(), false);
}
```

Mouse events for the content part of the scroll view can be handled by reimplementing special event handlers in `QScrollView`, whose names all start with `contents`. Behind the scenes, `QScrollView` automatically converts the viewport coordinates to content coordinates, so we don't need to convert them ourselves.

```
void ImageEditor::setImagePixel(const QPoint &pos, bool opaque)
{
    int i = pos.x() / zoom;
    int j = pos.y() / zoom;

    if (curImage.rect().contains(i, j)) {
        if (opaque)
            curImage.setPixel(i, j, penColor().rgb());
        else
            curImage.setPixel(i, j, qRgba(0, 0, 0, 0));

        QPainter painter(viewport());
        painter.translate(-contentsX(), -contentsY());
        drawImagePixel(&painter, i, j);
    }
}
```

The `setImagePixel()` function is called from `contentsMousePressEvent()` and `contentsMouseMoveEvent()` to set or clear a pixel. The code is almost the same as the original version, except for the way the `QPainter` object is initialized. We pass `viewport()` as the parent because the painting is performed on the

viewport, and we translate the QPainter's coordinate system to account for the scrolling offset.

We could replace the three lines that deal with the QPainter with this line:

```
updateContents(i * zoom, j * zoom, zoom, zoom);
```

This would tell QScrollView to update only the small rectangular area occupied by the (zoomed) image pixel. But since we didn't optimize drawContents() to draw only the necessary area, this would be inefficient, so it's better to construct a QPainter and do the painting ourselves.

If we use ImageEditor now, it is practically indistinguishable from the original, QWidget-based IconEditor used inside a QScrollView widget. However, for certain more sophisticated widgets, subclassing QScrollView is the more natural approach. For example, a class such as QTextEdit that implements word-wrapping needs tight integration between the document that is shown and the QScrollView.

Also note that you should subclass QScrollView if the contents are likely to be very tall or wide, because some window systems don't support widgets that are larger than 32,767 pixels.

One thing that the ImageEditor example doesn't demonstrate is that we can put child widgets in the viewport area. The child widgets simply need to be added using addWidget(), and can be moved using moveWidget(). Whenever the user scrolls the content area, QScrollView automatically moves the child widgets on screen. (If the QScrollView contains many child widgets, this can slow down scrolling. We can call enableClipper(true) to optimize this case.) One example where this approach would make sense is for a web browser. Most of the contents would be drawn directly on the viewport, but buttons and other form-entry elements would be represented by child widgets.

Dock Windows

Dock windows are windows that can be docked in dock areas. Toolbars are the primary example of dock windows, but there can be other types.

QMainWindow provides four dock areas: one above, one below, one to the left, and one to the right of the window's central widget. When we create QToolBars, they automatically put themselves in their parent's top dock area.

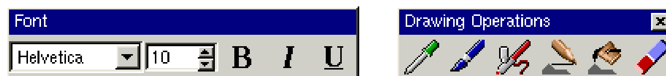

Figure 6.11. Floating dock windows

Every dock window has a handle. This appears as two gray lines at the left or top of each dock window shown in Figure 6.12. Users can move dock windows from one dock area to another by dragging the handle. They can also detach a

dock window from an area and let the dock window float as a top-level window by dragging the dock window outside of any dock area. Free floating dock windows have their own caption, and can have a close button. They are always "on top" of their main window.

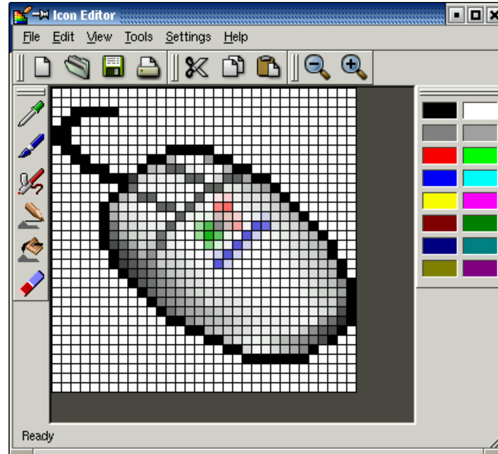

Figure 6.12. A `QMainWindow` with five dock windows

To turn on the close button when the dock window is floating, call `setCloseMode()` as follows:

```
dockWindow->setCloseMode(QDockWindow::Undocked);
```

`QDockArea` provides a context menu with the list of all dock windows and toolbars. Once a dock window is closed, the user can restore it using the context menu.

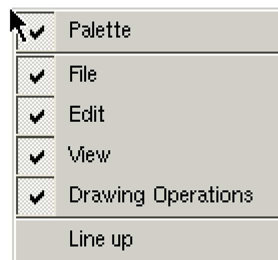

Figure 6.13. A `QDockArea` context menu

Dock windows must be subclasses of `QDockWindow`. If we just need a toolbar with buttons and some other widgets, we can use `QToolBar`, which inherits `QDockWindow`. Here's how to create a `QToolBar` containing a `QComboBox`, a `QSpinBox`, and some toolbar buttons, and how to put it in the bottom dock area:

```
QToolBar *toolBar = new QToolBar(tr("Font"), this);
QComboBox *fontComboBox = new QComboBox(true, toolBar);
```

```
QSpinBox *fontSize = new QSpinBox(toolBar);
boldAct->addTo(toolBar);
italicAct->addTo(toolBar);
underlineAct->addTo(toolBar);
moveDockWindow(toolBar, DockBottom);
```

This toolbar would look ugly if the user moves it to a `QMainWindow`'s left or right dock areas because the `QComboBox` and the `QSpinBox` require too much horizontal space. To prevent this from happening, we can call `QMainWindow::setDockEnabled()` as follows:

```
setDockEnabled(toolBar, DockLeft, false);
setDockEnabled(toolBar, DockRight, false);
```

If what we need is something more like a floating widget or tool palette, we can use `QDockWindow` directly, by calling `setWidget()` to set the widget to be shown inside the `QDockWindow`. The widget can be as complicated as we like. If we want the user to be able to resize the dock window even when it's in a dock area, we can call `setResizeEnabled()` on the dock window. The dock window will then be rendered with a splitter-like handle on the side.

If we want the widget to change itself depending on whether it is put in a horizontal or in a vertical dock area, we can reimplement `QDockWindow::setOrientation()` and change it there.

If we want to save the position of all the toolbars and other dock windows so that we can restore them the next time the application is run, we can write code that is similar to the code we used to save a `QSplitter`'s state (p. 143), using `QMainWindow`'s `<<` operator to write out the state and `QMainWindow`'s `>>` operator to read it back in.

Applications like Microsoft Visual Studio and *Qt Designer* make extensive use of dock windows to provide a very flexible user interface. In Qt, this kind of user interface is usually achieved by using a `QMainWindow` with many custom `QDockWindow`s and a `QWorkspace` in the middle to control MDI child windows.

Multiple Document Interface

Applications that provide multiple documents within the main window's central area are called MDI (multiple document interface) applications. In Qt, an MDI application is created by using the `QWorkspace` class as the central widget and by making each document window a child of the `QWorkspace`.

It is conventional for MDI applications to provide a Windows menu that includes some commands for managing the windows and the list of windows. The active window is identified with a checkmark. The user can make any window active by clicking its entry in the Windows menu.

In this section, we will develop the Editor application shown in Figure 6.14 to demonstrate how to create an MDI application and how to implement its Windows menu.

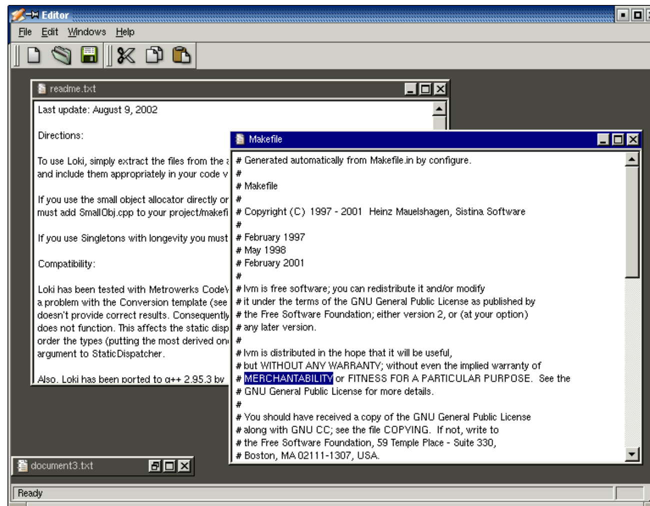

Figure 6.14. The Editor application

The application consists of two classes: `MainWindow` and `Editor`. Its code is on the CD, and since most of it is the same or similar to the Spreadsheet application from Part I, we will only present the new code.

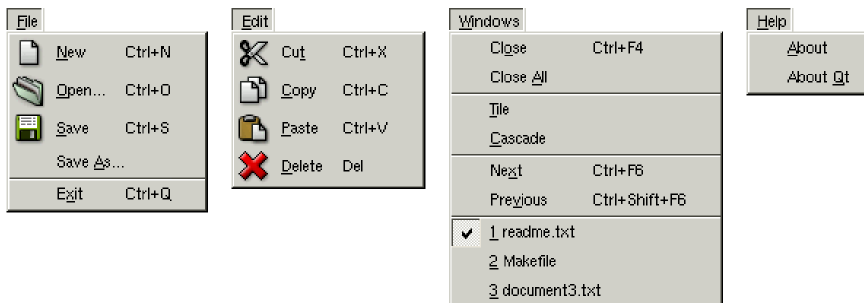

Figure 6.15. The Editor application's menus

Let's start with the `MainWindow` class.

```
MainWindow::MainWindow(QWidget *parent, const char *name)
    : QMainWindow(parent, name)
{
    workspace = new QWorkspace(this);
    setCentralWidget(workspace);
    connect(workspace, SIGNAL(windowActivated(QWidget *)),
            this, SLOT(updateMenus()));
    connect(workspace, SIGNAL(windowActivated(QWidget *)),
            this, SLOT(updateModIndicator()));

    createActions();
    createMenus();
    createToolBars();
```

```
    createStatusBar();

    setCaption(tr("Editor"));
    setIcon(QPixmap::fromMimeSource("icon.png"));
}
```

In the `MainWindow` constructor, we create a `QWorkspace` widget and make it the central widget. We connect the `QWorkspace`'s `windowActivated()` signal to two private slots. These slots ensure that the menus and the status bar always reflect the state of the currently active child window.

```
void MainWindow::newFile()
{
    Editor *editor = createEditor();
    editor->newFile();
    editor->show();
}
```

The `newFile()` slot corresponds to the File|New menu option. It depends on the `createEditor()` private function to create a child `Editor` window.

```
Editor *MainWindow::createEditor()
{
    Editor *editor = new Editor(workspace);
    connect(editor, SIGNAL(copyAvailable(bool)),
            this, SLOT(copyAvailable(bool)));
    connect(editor, SIGNAL(modificationChanged(bool)),
            this, SLOT(updateModIndicator()));
    return editor;
}
```

The `createEditor()` function creates an `Editor` widget and sets up two signal–slot connections. The first connection ensures that Edit|Cut and Edit|Copy are enabled or disabled depending on whether there is any selected text. The second connection ensures that the MOD indicator in the status bar is always up to date.

Because we are using MDI, it is possible that there will be multiple `Editor` widgets in use. This is a concern since we are only interested in responding to the `copyAvailable(bool)` and `modificationChanged()` signals from the active `Editor` window, not from the others. But these signals can only ever be emitted by the active window, so this isn't really a problem.

```
void MainWindow::open()
{
    Editor *editor = createEditor();
    if (editor->open())
        editor->show();
    else
        editor->close();
}
```

The `open()` function corresponds to File|Open. It creates a new `Editor` for the new document and calls `open()` on the `Editor`. It makes more sense to implement the file operations in the `Editor` class than in the `MainWindow` class, be-

cause each `Editor` needs to maintain its own independent state. If the `open()` fails, we simply close the editor since the user will have already been notified of the error.

```
void MainWindow::save()
{
    if (activeEditor()) {
        activeEditor()->save();
        updateModIndicator();
    }
}
```

The `save()` slot calls `save()` on the active editor, if there is one. Again, the code that performs the real work is located in the `Editor` class.

```
Editor *MainWindow::activeEditor()
{
    return (Editor *)workspace->activeWindow();
}
```

The `activeEditor()` private function returns the active child window as an `Editor` pointer.

```
void MainWindow::cut()
{
    if (activeEditor())
        activeEditor()->cut();
}
```

The `cut()` slot calls `cut()` on the active editor. The `copy()`, `paste()`, and `del()` slots follow the same pattern.

```
void MainWindow::updateMenus()
{
    bool hasEditor = (activeEditor() != 0);
    saveAct->setEnabled(hasEditor);
    saveAsAct->setEnabled(hasEditor);
    pasteAct->setEnabled(hasEditor);
    deleteAct->setEnabled(hasEditor);
    copyAvailable(activeEditor()
                  && activeEditor()->hasSelectedText());
    closeAct->setEnabled(hasEditor);
    closeAllAct->setEnabled(hasEditor);
    tileAct->setEnabled(hasEditor);
    cascadeAct->setEnabled(hasEditor);
    nextAct->setEnabled(hasEditor);
    previousAct->setEnabled(hasEditor);

    windowsMenu->clear();
    createWindowsMenu();
}
```

The `updateMenus()` slot is called whenever a window is activated (or when the last window is closed) to update the menu system, thanks to the signal–slot connection we put in the `MainWindow` constructor.

Most menu options only make sense if there is an active window, so we disable them if there isn't one. Then we clear the Windows menu and call `createWindowsMenu()` to reinitialize it with a fresh list of child windows.

```
void MainWindow::createWindowsMenu()
{
    closeAct->addTo(windowsMenu);
    closeAllAct->addTo(windowsMenu);
    windowsMenu->insertSeparator();
    tileAct->addTo(windowsMenu);
    cascadeAct->addTo(windowsMenu);
    windowsMenu->insertSeparator();
    nextAct->addTo(windowsMenu);
    previousAct->addTo(windowsMenu);

    if (activeEditor()) {
        windowsMenu->insertSeparator();
        windows = workspace->windowList();
        int numVisibleEditors = 0;

        for (int i = 0; i < (int)windows.count(); ++i) {
            QWidget *win = windows.at(i);
            if (!win->isHidden()) {
                QString text = tr("%1 %2")
                               .arg(numVisibleEditors + 1)
                               .arg(win->caption());
                if (numVisibleEditors < 9)
                    text.prepend("&");
                int id = windowsMenu->insertItem(
                              text, this, SLOT(activateWindow(int)));
                bool isActive = (activeEditor() == win);
                windowsMenu->setItemChecked(id, isActive);
                windowsMenu->setItemParameter(id, i);
                ++numVisibleEditors;
            }
        }
    }
}
```

The `createWindowsMenu()` private function fills the Windows menu with actions and a list of visible windows. The actions are all typical of such menus and are easily implemented using `QWorkspace`'s `closeActiveWindow()`, `closeAllWindows()`, `tile()`, and `cascade()` slots.

The entry for the active window is shown with a checkmark next to its name. When the user chooses a window entry, the `activateWindow()` slot is called with the index in the `windows` list as the parameter, because of the call to `setItemParameter()`. This is very similar to what we did in Chapter 3 when we implemented the Spreadsheet application's recently opened files list (p. 54).

For the first nine entries, we put an ampersand in front of the number to make that number's single digit into a shortcut key. We don't provide a shortcut key for the other entries.

```
void MainWindow::activateWindow(int param)
{
    QWidget *win = windows.at(param);
    win->show();
    win->setFocus();
}
```

The `activateWindow()` function is called when a window is chosen from the Windows menu. The `int` parameter is the value that we set with `setItemParameter()`. The `windows` data member holds the list of windows and was set in `createWindowsMenu()`.

```
void MainWindow::copyAvailable(bool available)
{
    cutAct->setEnabled(available);
    copyAct->setEnabled(available);
}
```

The `copyAvailable()` slot is called whenever text is selected or deselected in an editor. It is also called from `updateMenus()`. It enables or disables the Cut and Copy actions.

```
void MainWindow::updateModIndicator()
{
    if (activeEditor() && activeEditor()->isModified())
        modLabel->setText(tr("MOD"));
    else
        modLabel->clear();
}
```

The `updateModIndicator()` updates the MOD indicator in the status bar. It is called whenever text is modified in an editor. It is also called when a new window is activated.

```
void MainWindow::closeEvent(QCloseEvent *event)
{
    workspace->closeAllWindows();
    if (activeEditor())
        event->ignore();
    else
        event->accept();
}
```

The `closeEvent()` function is reimplemented to close all child windows. If one of the child widgets "ignores" its close event (presumably because the user canceled an "unsaved changes" message box), we ignore the close event for the `MainWindow`; otherwise we accept it, resulting in Qt closing the window. If we didn't reimplement `closeEvent()` in `MainWindow`, the user would not be given the opportunity to save any unsaved changes.

We have now finished our review of `MainWindow`, so we can move on to the `Editor` implementation. The `Editor` class represents one child window. It inherits from `QTextEdit`, which provides the text editing functionality. Just as any Qt widget can be used as a stand-alone window, any Qt widget can be used as a child window in an MDI workspace.

Here's the class definition:

```
class Editor : public QTextEdit
{
    Q_OBJECT
public:
    Editor(QWidget *parent = 0, const char *name = 0);

    void newFile();
    bool open();
    bool openFile(const QString &fileName);
    bool save();
    bool saveAs();
    QSize sizeHint() const;

signals:
    void message(const QString &fileName, int delay);

protected:
    void closeEvent(QCloseEvent *event);

private:
    bool maybeSave();
    void saveFile(const QString &fileName);
    void setCurrentFile(const QString &fileName);
    QString strippedName(const QString &fullFileName);
    bool readFile(const QString &fileName);
    bool writeFile(const QString &fileName);

    QString curFile;
    bool isUntitled;
    QString fileFilters;
};
```

Four of the private functions that were in the Spreadsheet application's `MainWindow` class (p. 51) are also present in the `Editor` class: `maybeSave()`, `saveFile()`, `setCurrentFile()`, and `strippedName()`.

```
Editor::Editor(QWidget *parent, const char *name)
    : QTextEdit(parent, name)
{
    setWFlags(WDestructiveClose);
    setIcon(QPixmap::fromMimeSource("document.png"));

    isUntitled = true;
    fileFilters = tr("Text files (*.txt)\n"
                     "All files (*)");
}
```

The `Editor` constructor sets the `WDestructiveClose` flag using `setWFlags()`. When a class constructor doesn't provide a `flags` parameter (as is the case with `QTextEdit`), we can still set most flags using `setWFlags()`.

Since we allow users to create any number of editor windows, we must make some provision for naming them so that they can be distinguished before they have been saved for the first time. One common way of handling this is to allocate names that include a number (for example, `document1.txt`). We use the

`isUntitled` variable to distinguish between names supplied by the user and names we have created programmatically.

After the constructor, we expect either `newFile()` or `open()` to be called.

```
void Editor::newFile()
{
    static int documentNumber = 1;

    curFile = tr("document%1.txt").arg(documentNumber);
    setCaption(curFile);
    isUntitled = true;
    ++documentNumber;
}
```

The `newFile()` function generates a name like `document2.txt` for the new document. The code belongs in `newFile()`, rather than the constructor, because we don't want to consume numbers when we call `open()` to open an existing document in a newly created `Editor`. Since `documentNumber` is declared static, it is shared across all `Editor` instances.

```
bool Editor::open()
{
    QString fileName =
            QFileDialog::getOpenFileName(".", fileFilters, this);
    if (fileName.isEmpty())
        return false;

    return openFile(fileName);
}
```

The `open()` function tries to open an existing file using `openFile()`.

```
bool Editor::save()
{
    if (isUntitled) {
        return saveAs();
    } else {
        saveFile(curFile);
        return true;
    }
}
```

The `save()` function uses the `isUntitled` variable to determine whether it should call `saveFile()` or `saveAs()`.

```
void Editor::closeEvent(QCloseEvent *event)
{
    if (maybeSave())
        event->accept();
    else
        event->ignore();
}
```

The `closeEvent()` function is reimplemented to allow the user to save unsaved changes. The logic is coded in the `maybeSave()` function, which pops up a message box that asks, "Do you want to save your changes?" If `maybeSave()`

returns `true`, we accept the close event; otherwise, we "ignore" it and leave the window unaffected by it.

```
void Editor::setCurrentFile(const QString &fileName)
{
    curFile = fileName;
    setCaption(strippedName(curFile));
    isUntitled = false;
    setModified(false);
}
```

The `setCurrentFile()` function is called from `openFile()` and `saveFile()` to update the `curFile` and `isUntitled` variables, to set the window caption, and to set the editor's "modified" flag to `false`. The `Editor` class inherits `setModified()` and `isModified()` from `QTextEdit`, so it doesn't need to maintain its own modified flag. Whenever the user modifies the text in the editor, `QTextEdit` emits the `modificationChanged()` signal and sets its internal modified flag to `true`.

```
QSize Editor::sizeHint() const
{
    return QSize(72 * fontMetrics().width('x'),
                 25 * fontMetrics().lineSpacing());
}
```

The `sizeHint()` function returns a size based on the width of the letter 'x' and the height of a text line. `QWorkspace` uses the size hint to give an initial size to the window.

Finally, here's the Editor application's `main.cpp` file:

```
#include <qapplication.h>

#include "mainwindow.h"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    MainWindow mainWin;
    app.setMainWidget(&mainWin);

    if (argc > 1) {
        for (int i = 1; i < argc; ++i)
            mainWin.openFile(argv[i]);
    } else {
        mainWin.newFile();
    }

    mainWin.show();
    return app.exec();
}
```

If the user specifies any files on the command line, we attempt to load them. Otherwise, we start with an empty document. Qt-specific command-line options, such as `-style` and `-font`, are automatically removed from the argument list by the `QApplication` constructor. So if we write

```
editor -style=motif readme.txt
```

on the command line, the Editor application starts up with one document, `readme.txt`.

MDI is one way of handling multiple documents simultaneously. Another approach is to use multiple top-level windows. This approach is covered in the "Multiple Documents" section of Chapter 3.