Nuevas Tecnologías de la Programación

Módulo 2: Programación gráfica en entorno UNIX con una librería de alto nivel (Qt)

Andrés Cano Utrera
Dpto. Ciencias de la Computación e I.A
Universidad de Granada

Noviembre de 2007

Índice

1. Introducción a la programación con Qt

- Qt es un toolkit de plataforma cruzada (*cross-platform*) desarrollado por *Trolltech* (empresa Noruega) para construir GUIs en C++
- Puede funcionar en varias plataformas: Microsoft Windows, Unix/X11, Mac OS X
- Las primeras versiones datan de 1996
- Con él se han desarrollado aplicaciones tales como: Google Earth, Adobe Photoshop Elements, Skype y KDE
- Se distribuye bajo ediciones comerciales o bien Open Source.
- Incluye una extensa librería de clases C++ y utilidades para construir las aplicaciones de forma rápida y sencilla.

2. Primeros programas con Qt

2.1. Primer programa con Qt

Hello Qt: chap01/hello.cpp

```
1  #include <QApplication>
2  #include <QLabel>
3  int main(int argc, char *argv[])
4  {
5      QApplication app(argc, argv);
6      QLabel *label = new QLabel("Hello Qt!");
7      label->show();
8      return app.exec();
9  }
```

- Las líneas 1 y 2 incluyen las definiciones de las clases `QApplication` y `QLabel`.

  Para cada clase Qt existe un fichero de cabecera con el mismo nombre

- La línea 5 crea un objeto `QApplication`
  - El constructor necesita `argc` y `argv` pues Qt soporta algunos argumentos propios en la línea de comandos.
- La línea 6 crea un *widget* (window gadget) (control o componente y contenedor al mismo tiempo) `QLabel` con la cadena ‘‘`Hello Qt!`’’
  - Los widgets pueden contener otros widgets.
  - Normalmente las aplicaciones usan `QMainWindow` o `QDialog` como ventana de la aplicación.
  - Por ejemplo, la ventana de la aplicación contiene normalmente un `QMenuBar`, `QToolBars`, una `QStatusBar` y otros widgets.
  - Pero cualquier widget puede actuar como ventana como ocurre en el ejemplo con `QLabel`.

- La línea 7 muestra la etiqueta en pantalla.

  Los widgets se crean siempre ocultos, para que puedan optimizarse antes de mostrarlos (evitando el parpadeo).

- La línea 8 da el control de la aplicación a Qt, comenzando el bucle de eventos.

2.2. Compilación del programa

- El código lo incluimos en un fichero `hello.cpp` dentro del directorio `hello`.
- Dentro del directorio `hello` ejecutamos para crear el *fichero de proyecto* (`hello.pro`):

  ```
  qmake-q4 -project
  ```

- Para crear el fichero `Makefile`:

  ```
  qmake-q4 hello.pro
  ```

- Para obtener el ejecutable:

  ```
  make
  ```

2.3. Conexión de acciones

Ejemplo de control de eventos: chap01/quit.cpp

```
1  #include <QApplication>
2  #include <QPushButton>
3  int main(int argc, char *argv[])
4  {
5      QApplication app(argc, argv);
6      QPushButton *button = new QPushButton("Quit");
7      QObject::connect(button, SIGNAL(clicked()),
8                       &app, SLOT(quit()));
9      button->show();
10     return app.exec();
11 }
```

- Los widgets emiten señales (*signals*) para indicar que se ha producido una acción de usuario o cambio de estado en el widget.
- Las señales pueden conectarse a una función (*slot*) que se ejecutará automáticamente cuando se emita la señal.
- En la línea 7 se conecta la señal `clicked()` del botón al slot `quit()` del objeto `QApplication`.

2.4. Gestores de posicionamiento

- Permiten controlar automáticamente la geometría (posición y tamaño) de los widgets que contiene.
  - `QHBoxLayout`: Coloca los widgets horizontalmente de izquierda a derecha.
  - `QVBoxLayout`: Coloca los widgets verticalmente de arriba hacia abajo.
  - `QGridLayout`: Coloca los widgets en una rejilla.

- El siguiente programa usa tres widgets: `QSpinBox`, `QSlider` y `QWidget` (ventana principal).
- Los dos primeros serán hijos del tercero.

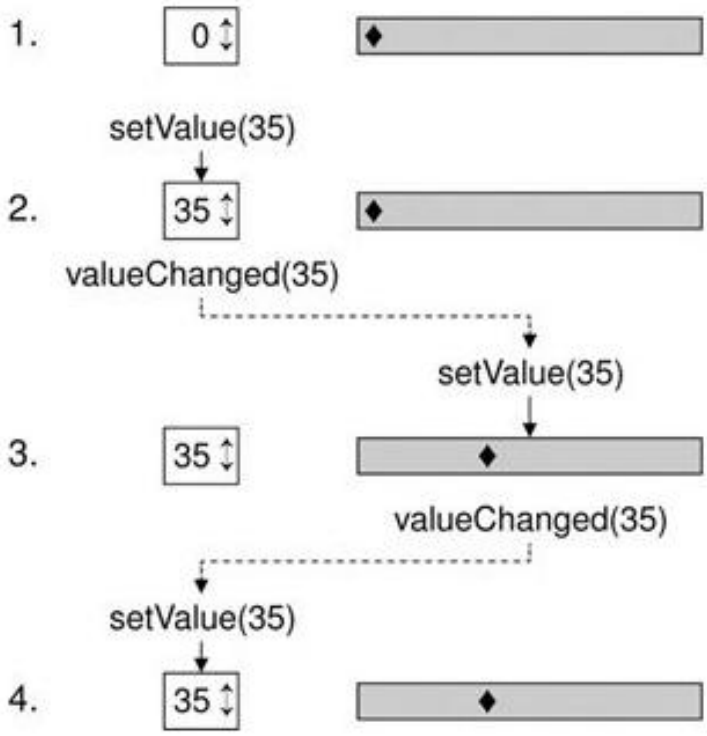
Ejemplo de Layouts y control de eventos: chap01/age.cpp

```
1  #include <QApplication>
2  #include <QHBoxLayout>
3  #include <QSlider>
4  #include <QSpinBox>
5  int main(int argc, char *argv[])
6  {
7      QApplication app(argc, argv);
8      QWidget *window = new QWidget;
9      window->setWindowTitle("Enter Your Age");
10     QSpinBox *spinBox = new QSpinBox;
11     QSlider *slider = new QSlider(Qt::Horizontal);
12     spinBox->setRange(0, 130);
13     slider->setRange(0, 130);
14     QObject::connect(spinBox, SIGNAL(valueChanged(int)),
15                      slider, SLOT(setValue(int)));
16     QObject::connect(slider, SIGNAL(valueChanged(int)),
17                      spinBox, SLOT(setValue(int)));
18     spinBox->setValue(35);
19     QHBoxLayout *layout = new QHBoxLayout;
20     layout->addWidget(spinBox);
21     layout->addWidget(slider);
22     window->setLayout(layout);
23     window->show();
24     return app.exec();
25 }
```

- Las líneas 8 a 11 crean los tres widgets.

  Al crear un widget debe pasarse el padre en el constructor, aunque en el ejemplo no hace falta hacerlo por lo que veremos más adelante.

- Las líneas 12 y 13 establecen los rangos válidos para el `QSpinBox` y `QSlider`.
- Las líneas 14 a 17 hacen que los objetos `QSpinBox` y `QSlider` estén siempre sincronizados mostrando el mismo valor.



- Las líneas 19 a 22 colocan los widgets `QSpinBox` y `QSlider` usando un gestor de posicionamiento (*layout manager*)
- La línea 22, a parte de instalar el gestor de posicionamiento en la ventana, hace que los widgets `QSpinBox` y `QSlider` se hagan hijos del widget de la ventana (`window`)

3. Creación de diálogos

- La mayoría de las aplicaciones contienen una ventana principal con una barra de menús, una barra de herramientas, y varios diálogos.
- Los *diálogos* muestran al usuario una serie de opciones que el usuario puede modificar de valor.
- Qt dispone de una serie de diálogos predefinidos.
- También permite crear nuestros propios diálogos.
- En Qt podemos crear tanto diálogos *modales* como *no modales* con una clase que herede de `QDialog`:
  - **Diálogos modales**: Bloquean el resto del interfaz gráfico hasta que el usuario los cierra.
    - Se muestran con el método `exec()` (que devuelve un entero que indica que el diálogo se ha cerrado).
    - `exec()` devuelve habitualmente los valores `QDialog:Accepted` o `QDialog:Rejected` (correspondientes a la pulsación de los botones **OK** o **Cancel**.
    - Normalmente se conectan los botones del diálogo a los slots `QDialog:accept()`, `QDialog:reject()` o `QDialog:done(int)` para que `exec()` devuelva el valor correspondiente.

Ejemplo de diálogo modal: MisSources/modal/modal.cpp

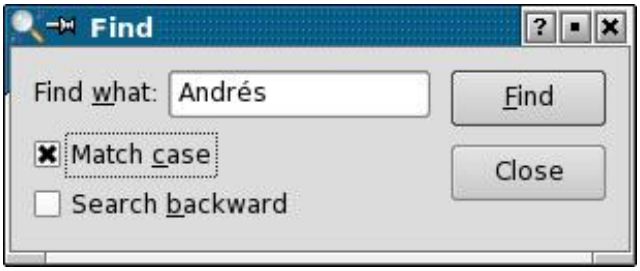
```
1  class MyDialog : public QDialog
2  {
3    Q_OBJECT
4  public:
5    MyDialog();
6    bool isCheckBoxChecked() const {
7            return _checkbox->isChecked(); }
8  private:
9    QCheckBox* _checkbox;
10 };
```

```
11  MyDialog::MyDialog() :
12    QDialog( 0)
13  {
14    _checkbox = new QCheckBox( "Check me", this );
15    QPushButton* okbutton = new QPushButton( "OK", this );
16    QObject::connect( okbutton, SIGNAL( clicked() ),
17                      this, SLOT( accept() ) );
18    QPushButton* cancelbutton = new QPushButton( "Cancel",
19                                                 this );
20    QObject::connect( cancelbutton, SIGNAL( clicked() ),
21                      this, SLOT( reject() ) );
22  }
23
24  class ClickReceiver : public QObject
25  {
26    Q_OBJECT
27  public slots:
28    void slotShowDialog();
29  };
30  void ClickReceiver::slotShowDialog()
31  {
32    MyDialog* dialog = new MyDialog();
33    int ret = dialog->exec();
34    if( ret == QDialog::Accepted )
35      {
36        qDebug( "User pressed OK; check box was %s checked.\n ",
37          (dialog->isCheckBoxChecked() ? "" : "not") );
38      }
39    else
40      {
41        qDebug( "User pressed Cancel\n" );
42      }
43  }
```

- **Diálogos no modales**: No bloquean el resto del interfaz gráfico.

3.1. Ejemplo de creación de una subclase de Diálogo

- Crearemos una subclase independiente por sí sola, con sus propias señales y slots.
- El código fuente lo dividimos en dos ficheros: `finddialog.h` y `finddialog.cpp`.



Ejemplo de diálogo: chap02/find/finddialog.h

```
1  #ifndef FINDDIALOG_H
2  #define FINDDIALOG_H
3  #include <QDialog>
4  class QCheckBox;
5  class QLabel;
6  class QLineEdit;
7  class QPushButton;
```

- Las líneas 1, 2 y 27 impiden que el fichero de cabecera pueda ser incluido varias veces.
- La línea 3 incluye la clase base para los diálogos Qt: `QDialog` (que es subclase de `QWidget`).
- Las líneas 4 a 7 son **declaraciones adelantadas** (forward declarations) de las clases que se usarán más adelante.

```
8  class FindDialog : public QDialog
9  {
10     Q_OBJECT
11 public:
12     FindDialog(QWidget *parent = 0);
```

- La línea 8 define la clase `FindDialog` como subclase de `QDialog`:
- En la línea 10 se incluye la macro `Q_OBJECT`, que es necesaria en todas las clases que definen sus propias señales o slots.
  - Esto hará que el makefile generado por `qmake` incluya reglas especiales para ejecutar `moc` (el compilador de meta-objetos)
  - Para que `moc` funcione correctamente, es necesario que pongamos la definición de la clase en un fichero `.h` separado del fichero de implementación.
  - `moc` genera código C++ que incluye este fichero `.h`.
- La definición del constructor de `FindDialog` es típico en clases de widgets Qt: `parent` especifica el widget padre.
- `parent` es por defecto un puntero nulo, que significa que el diálogo no tiene ningún padre.

```
13 signals:
14     void findNext(const QString &str, Qt::CaseSensitivity cs);
15     void findPrevious(const QString &str, Qt::CaseSensitivity cs);
```

- La sección `signals` de la línea 13 a 15 declara dos señales que el diálogo emite cuando el usuario pincha el botón *Find*
  - Si está seleccionada la opción *Search backward*, se emite `findPrevious()`
  - En caso contrario se emite `findNext()`
- La palabra `signals` es una macro que el preprocesador de C++ convertirá en código C++.
- `Qt::CaseSensitivity` es un tipo enumerado (`enum`) que puede tomar los valores `Qt::CaseSensitive` y `Qt::CaseInsensitive`

```
16 private slots:
17     void findClicked();
18     void enableFindButton(const QString &text);
19 private:
20     QLabel *label;
21     QLineEdit *lineEdit;
22     QCheckBox *caseCheckBox;
23     QCheckBox *backwardCheckBox;
24     QPushButton *findButton;
25     QPushButton *closeButton;
26 };
27 #endif
```

- En las líneas 16 a 18 se declaran dos slots (en la sección privada de la clase). La palabra `slots` es también una macro.
- En las líneas 19 a 25 se declaran punteros a los widgets hijos del diálogo para usarlos luego en los slots.

Ejemplo de diálogo: chap02/find/finddialog.cpp

```
1 #include <QtGui>
2 #include "finddialog.h"
```

- Qt contiene varios módulos, cada uno en su propia librería: QtCore, QtGui, QtNetwork, QtOpenGL, QtSql, QtSvg, QtXml, etc.
- El fichero de cabecera `<QtGui>` contiene la definición de todas las clases de QtCore y QtGui. Su inclusión evita tener que incluir cada clase individualmente.

```
3  FindDialog::FindDialog(QWidget *parent)
4      : QDialog(parent)
5  {
6      label = new QLabel(tr("Find &what:"));
7      lineEdit = new QLineEdit;
8      label->setBuddy(lineEdit);
9      caseCheckBox = new QCheckBox(tr("Match &case"));
10     backwardCheckBox = new QCheckBox(tr("Search &backward"));
11     findButton = new QPushButton(tr("&Find"));
12     findButton->setDefault(true);
13     findButton->setEnabled(false);
14     closeButton = new QPushButton(tr("Close"));
```

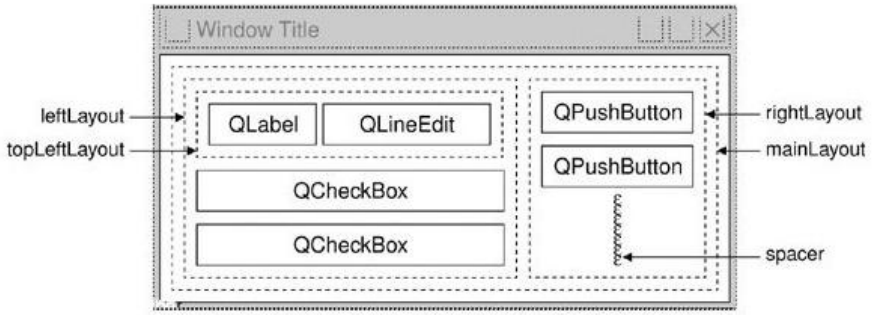
- En la línea 4 se pasa el parámetro `parent` al constructor de la clase base.
- La función `tr(cadena)` marca la cadena para ser traducida al lenguaje nativo. (está declarada en la clase `QObject` y todas las clases que contengan la macro `Q_OBJECT`)
- En la cadena usada al crear algunos objetos (líneas 6, 9, 10, 11) se usa el caracter `'&'` para definir una tecla aceleradora: Alt+tecla.
- La línea 8 hace que `lineEdit` sea el *amigote* (*buddy)* de `label`: al pulsar Alt+W el foco del teclado lo obtiene `lineEdit`.
- En la línea 12, el botón **Find** se hace el botón por defecto (es aquel que se ejecuta al pulsar la tecla Enter).
- La línea 13 desactiva el botón **Find** (aparece en grisaceo y no responde a los clicks)

```
15      connect(lineEdit, SIGNAL(textChanged(const QString &)),
16              this, SLOT(enableFindButton(const QString &)));
17      connect(findButton, SIGNAL(clicked()),
18              this, SLOT(findClicked()));
19      connect(closeButton, SIGNAL(clicked()),
20              this, SLOT(close()));
```

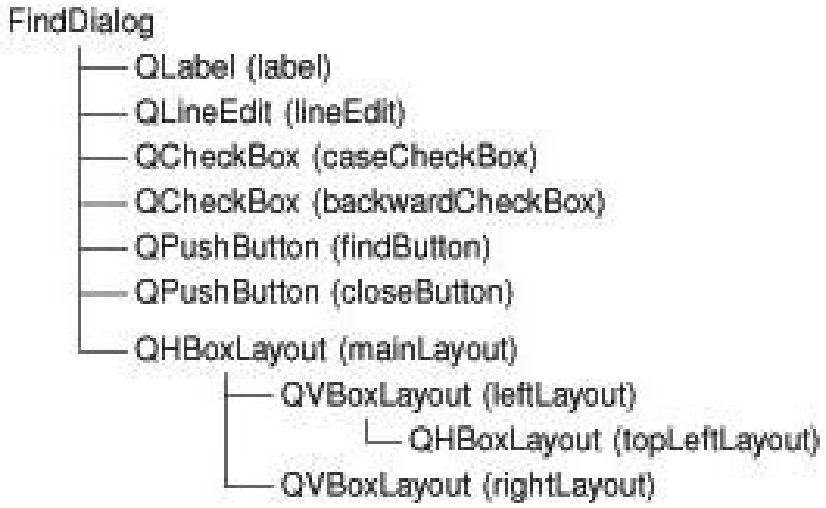
- Las líneas 15 a 20 conectan señales con slots:
  - Fijarse que no es necesario poner `QObject::` delante de las llamadas a `connect()` ya que `QObject` es una clase ancestral de `FindDialog`.
  - El slot `enableFindButton(const QString &)` se llama cuando cambia el texto de `lineEdit`.
  - El slot `findClicked(const QString &)` se llama cuando el usuario pincha el botón **Find**.
  - El slot `close()` (heredado de `QWidget`, oculta el widget sin borrarlo) se llama cuando el usuario pincha el botón **Close**.

```
21      QHBoxLayout *topLeftLayout = new QHBoxLayout;
22      topLeftLayout->addWidget(label);
23      topLeftLayout->addWidget(lineEdit);
24      QVBoxLayout *leftLayout = new QVBoxLayout;
25      leftLayout->addLayout(topLeftLayout);
26      leftLayout->addWidget(caseCheckBox);
27      leftLayout->addWidget(backwardCheckBox);
28      QVBoxLayout *rightLayout = new QVBoxLayout;
29      rightLayout->addWidget(findButton);
30      rightLayout->addWidget(closeButton);
31      rightLayout->addStretch();
32      QHBoxLayout *mainLayout = new QHBoxLayout;
33      mainLayout->addLayout(leftLayout);
34      mainLayout->addLayout(rightLayout);
35      setLayout(mainLayout);
36      setWindowTitle(tr("Find"));
```

- Las líneas 21 a 35 colocan los widgets hijos usando *gestores de posicionamiento*.
  - Layouts pueden contener widgets y otros layouts.
  - Combinando `QHBoxLayouts`, `QVBoxLayouts` y `QGridLayouts` podemos construir diálogos muy sofisticados.



  - Los layouts no son widgets (heredan de `QLayout` que a su vez hereda de `QObject`)
  - Al incluir un layout en otro (líneas 25, 33 y 34), los sublayouts se hacen sus hijos.
  - Al instalar en el diálogo (línea 35) el layout principal, éste y todos los widgets de la jerarquía de layouts se hacen hijos del diálogo




```
37      setFixedHeight(sizeHint().height());
38  }
```

- En la línea 37 establecemos un tamaño vertical fijo para el diálogo (`QWidget.sizeHint()` devuelve el tamaño ideal para un widget).
- No necesitamos añadir un destructor a `FindDialog` pues, aunque siempre se ha usado `new` para crear los widgets y layouts del diálogo, Qt automáticamente borra los objetos hijo cuando el padre se destruye.
  - Cuando se crea un objeto indicando el padre, el padre añade el objeto a su lista de hijos.
  - Si el padre es borrado, se borra automáticamente la lista de hijos, y así recursivamente.
  - Además, cuando el widget padre se borra, desaparece de pantalla éste, y los hijos que tenga.
  - Esto hace que en la práctica, no haga falta que borremos los objetos, salvo los creados con `new` que no tengan padre.

```
39 void FindDialog::findClicked()
40 {
41     QString text = lineEdit->text();
42     Qt::CaseSensitivity cs =
43             caseCheckBox->isChecked() ? Qt::CaseSensitive
44                                       : Qt::CaseInsensitive;
45     if (backwardCheckBox->isChecked()) {
46         emit findPrevious(text, cs);
47     } else {
48         emit findNext(text, cs);
49     }
50 }
51 void FindDialog::enableFindButton(const QString &text)
52 {
53     findButton->setEnabled(!text.isEmpty());
54 }
```

- El slot `findClicked()` (llamado al pinchar el botón **Find**) emite las señales `findPrevious()` o `findNext()`, dependiendo de si está o no seleccionado *Search backward*.
- La palabra `emit` es también propia de Qt, y convertida a código C++ por el preprocesador.
- El slot `enableFindButton()` (llamado cuando el usuario cambia el texto de `lineEdit`) activa el botón **Find** si hay algún texto en `lineEdit`

Ejemplo de diálogo: chap02/find/main.cpp

```
1 #include <QApplication>
2 #include "finddialog.h"
3 int main(int argc, char *argv[])
4 {
5     QApplication app(argc, argv);
6     FindDialog *dialog = new FindDialog;
7     dialog->show();
8     return app.exec();
9 }
```

3.2. Más sobre señales y slots

- La sentencia `connect()` tiene la siguiente forma:

  ```
  connect(sender, SIGNAL(signal), receiver, SLOT(slot));
  ```

  donde `sender` y `receiver` son punteros a `QObjects`.

- Las macros `SIGNAL()` y `SLOT()` convierten su argumento en un string.

- Una señal puede conectarse a varios slots:

  ```
  connect(slider, SIGNAL(valueChanged(int)),
          spinBox, SLOT(setValue(int)));
  connect(slider, SIGNAL(valueChanged(int)),
          this, SLOT(updateStatusBarIndicator(int)));
  ```

- Varias señales pueden conectarse al mismo slot:

  ```
  connect(lcd, SIGNAL(overflow()),
          this, SLOT(handleMathError()));
  connect(calculator, SIGNAL(divisionByZero()),
          this, SLOT(handleMathError()));
  ```

- Una señal puede conectarse a otra señal:

  ```
  connect(lineEdit, SIGNAL(textChanged(const QString &)),
          this, SIGNAL(updateRecord(const QString &)));
  ```

- Las conexiones pueden borrarse (aunque no se usa demasiado):

  ```
  disconnect(lcd, SIGNAL(overflow()), this, SLOT(handleMathError()));
  ```

- Para conectar una señal con un slot u otra señal, ambos deben tener los mismos tipos de parámetros y en el mismo orden:

  ```
  connect(ftp, SIGNAL(rawCommandReply(int, const QString &)),
          this, SLOT(processReply(int, const QString &)));
  ```

- Como excepción, si una señal tiene más parámetros que el slot, se ignoran los parámetros adicionales:

  ```
  connect(ftp, SIGNAL(rawCommandReply(int, const QString &)),
          this, SLOT(checkErrorCode(int)));
  ```

- No deben incluirse los nombres de los parámetros de las señales o slots al usar `connect`.

4. Un vistazo a los widgets de Qt

4.1. Botones

- Pueden etiquetarse con texto o icono (un pixmap)
- `QPushButton`: Representa un botón de comando (los normales).

  - Normalmente se conectan a un slot, cuando emiten la señal `clicked()`
  - También tienen disponibles las señales `pressed()`, `released()` y `toggled()`.
  - Se les puede asociar un menú popup con `setPopup()`
- `QToolButton`: También representa un botón de comando, aunque suelen usarse como botones *toggle*
  - Normalmente muestran una imagen (`QIcon`) y no una etiqueta.
  - Se crean normalmente al crear un `QAction` (objeto que luego puede incluirse en un menú y en la barra de herramientas)
- `QCheckBox` y `QRadioButton`

  - Los `QCheckBox` suelen aparecer en grupos donde podremos seleccionar cero, una o varias opciones.

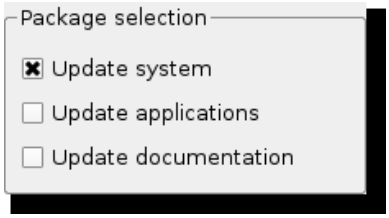
- Los `QRadioButton` también suelen colocarse en un grupo con varios `QRadioButton` donde podremos seleccionar exactamente uno.

```
QButtonGroup* bgroup = new QButtonGroup( "Heading", parent);
QRadioButton* radio1 = new QRadioButton( "Choice1", bgroup);
QRadioButton* radio2 = new QRadioButton( "Choice2", bgroup);
QRadioButton* radio3 = new QRadioButton( "Choice3", bgroup);
```

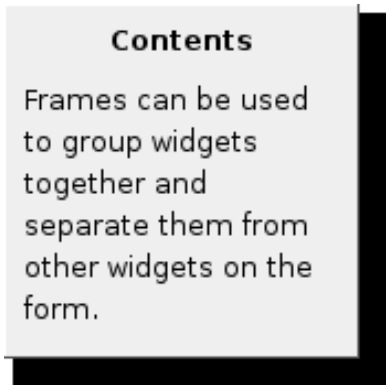
- Para comprobar si está seleccionado: `isChecked()`
- Para seleccionarlos: `setChecked()`
- Tienen disponibles las señales `clicked()`, `pressed()`, `released()` y `toggled()`.

4.2. Contenedores

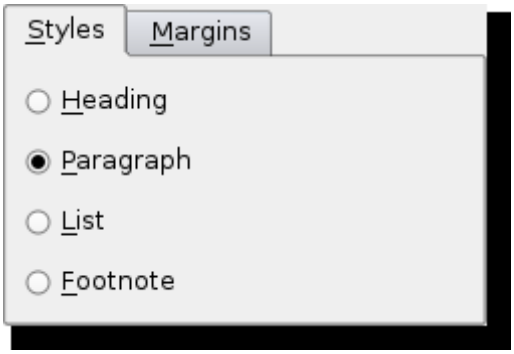
- Son widgets que contienen otros widgets:
  - `QGroupBox`



  - `QFrame`



- `QTabWidget`: Es un contenedor multipágina.



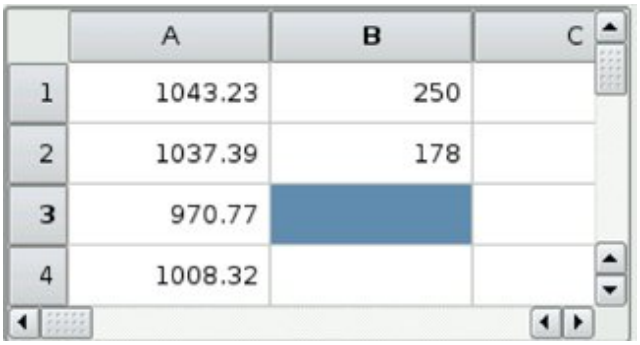
- `QToolBox`: Otro contenedor multipágina.

4.3. Visores de items

- `QListView` y `QListWidget`:

- `QTreeView`

- `QTableView`



4.4. Widgets para mostrar información

- `QLabel`: Permite mostrar texto simple, html o imágenes.

- `QProgressBar`

- `QLCDNumber`: Muestra un número con display LCD

- `QTextBrowser`: Es un subclase de `QTextEdit` y soporta html básico (listas, tablas, imágenes e hiperenlaces).

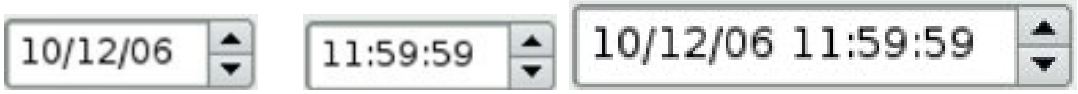
4.5. Widgets de entrada

- `QSpinBox`: Se usa para introducir valores enteros o de un conjunto discreto.

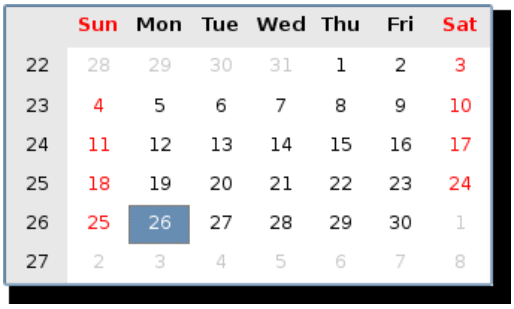
- `QDoubleSpinBox`:

- `QComboBox`: Es la combinación de un botón con una lista (popup)

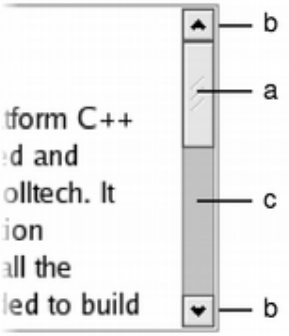
- `QDateEdit`, `QTimeEdit`, `QDateTimeEdit`: Permiten introducir fechas, la hora, o ambos simultaneamente.



- `QCalendarWidget`: Permite introducir fechas.

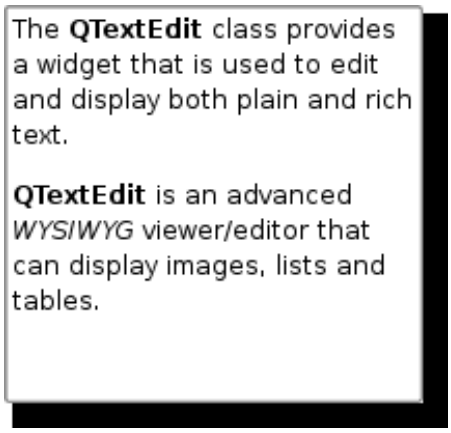


- `QScrollbar`: Barra de scroll horizontal o vertical
  - Permite acceder a partes de un documento que es mayor que el widget usado para mostrarlo.

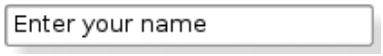


- `QSlider`: Deslizador (*slider*) horizontal o vertical
  - Permite mover la barra para traducir su posición a un rango determinado de valores enteros.

- `QTextEdit`: Es un visor/editor WYSIWYG que soporta texto enriquecido usando etiquetas HTML.

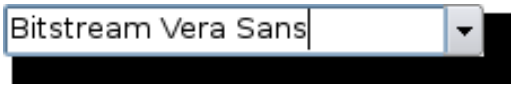


- `QLineEdit`: Editor de texto de una sola línea.



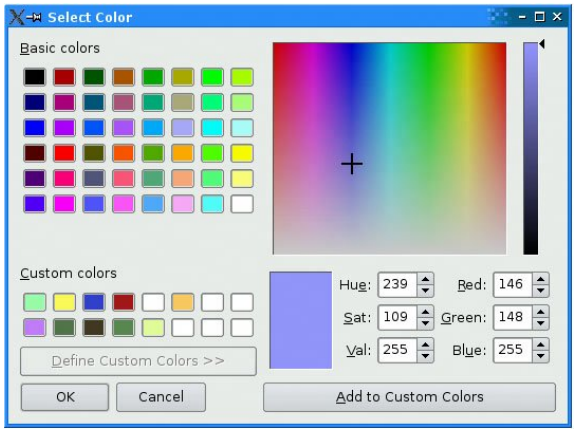
- `QDial`: Permite dar un valor en un rango determinado (por ejemplo grados 0-360).

`QFontComboBox`: Permite introducir un font

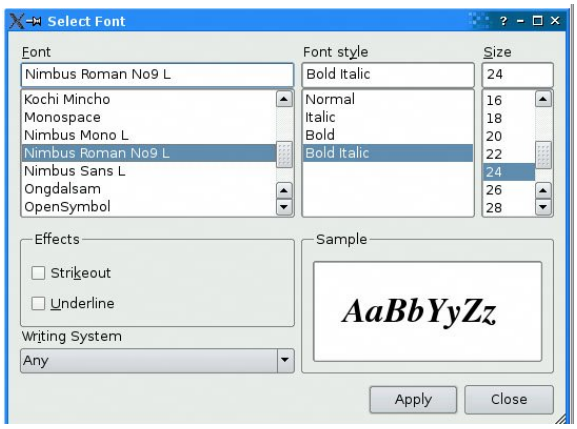


4.6. Diálogos predefinidos

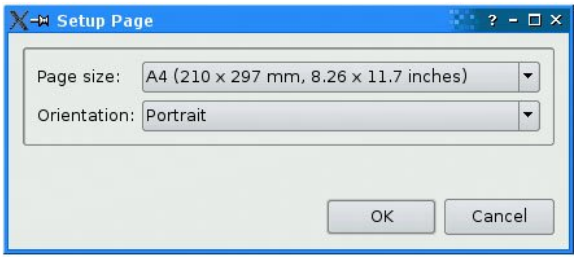
- `QColorDialog`



- `QFontDialog`



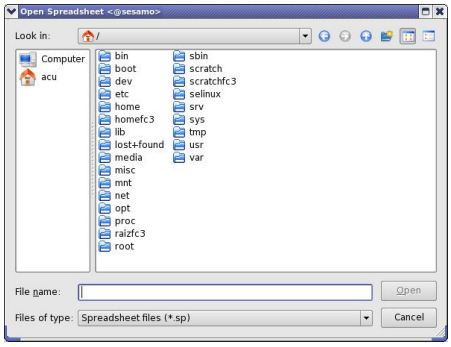
- `QPageSetupDialog`

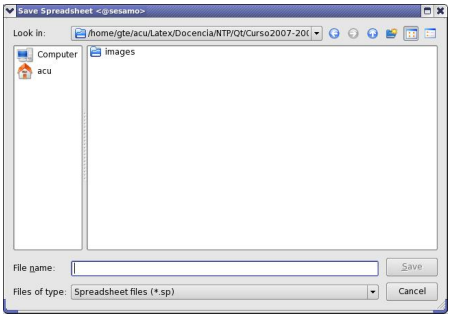


- `QFileDialog`

Ejemplo de QFileDialog:
chap03/spreadsheet/mainwindow.cpp

```
void MainWindow::open()
{
    if (okToContinue()) {
      QString fileName = QFileDialog::getOpenFileName(this,
          tr("Open Spreadsheet"), ".",
          tr("Spreadsheet files (*.sp)"));
      if (!fileName.isEmpty())
          loadFile(fileName);
    }
}
bool MainWindow::saveAs()
{
    QString fileName = QFileDialog::getSaveFileName(this,
      tr("Save Spreadsheet"), ".",
      tr("Spreadsheet files (*.sp)"));
    if (fileName.isEmpty())
      return false;
    return saveFile(fileName);
}
```

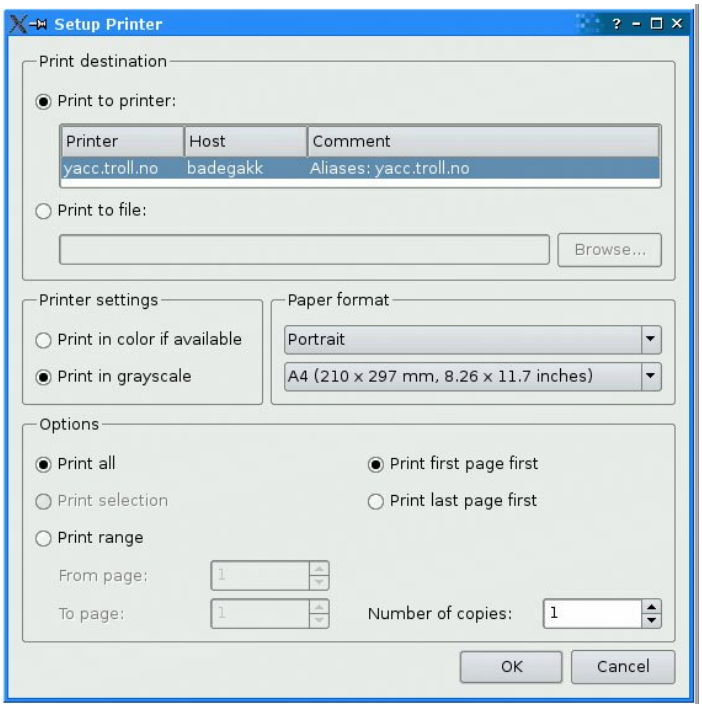




- Los argumentos de `getOpenFileName()` y `getSaveFileName()` son:
  - **Widget padre**: Los diálogos siempre son ventanas de alto nivel, pero si tienen padre, al mostrarse aparecen centrados encima del padre.
  - **Título**
  - **Directorio inicial**
  - **Filtros**: Texto descriptivo y un patrón. Ejemplo:

```
tr("Spreadsheet files (*.sp)\n"
   "Comma-separated values files (*.csv)\n"
   "Lotus 1-2-3 files (*.wk1 *.wks)")
```

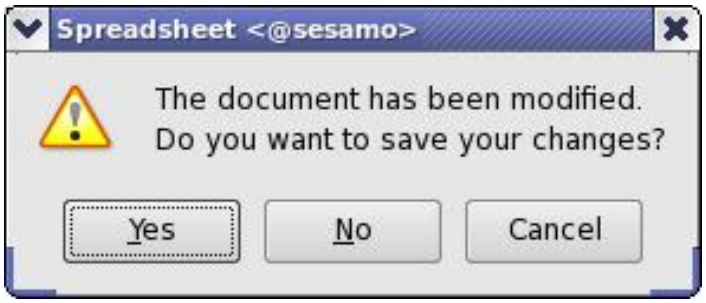
- `QPrintDialog`



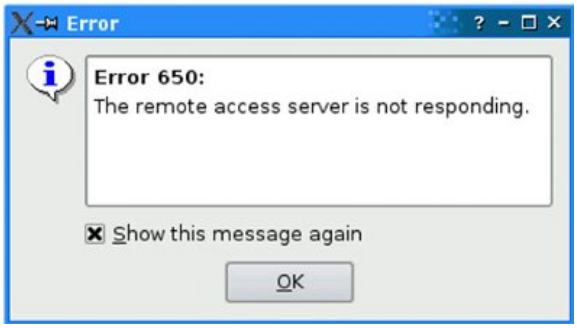
- `QMessageBox`

Ejemplo de QMessageBox: chap03/spreadsheet/mainwindow.cpp

```
bool MainWindow::okToContinue()
{
    if (isWindowModified()) {
        int r = QMessageBox::warning(this, tr("Spreadsheet"),
                    tr("The document has been modified.\n"
                       "Do you want to save your changes?"),
                    QMessageBox::Yes | QMessageBox::Default,
                    QMessageBox::No,
                    QMessageBox::Cancel | QMessageBox::Escape);
        if (r == QMessageBox::Yes) {
            return save();
        } else if (r == QMessageBox::Cancel) {
            return false;
        }
    }
    return true;
}
```

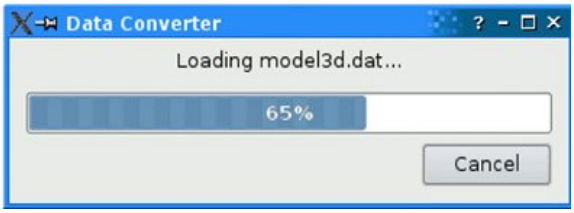


- `QErrorMessage`



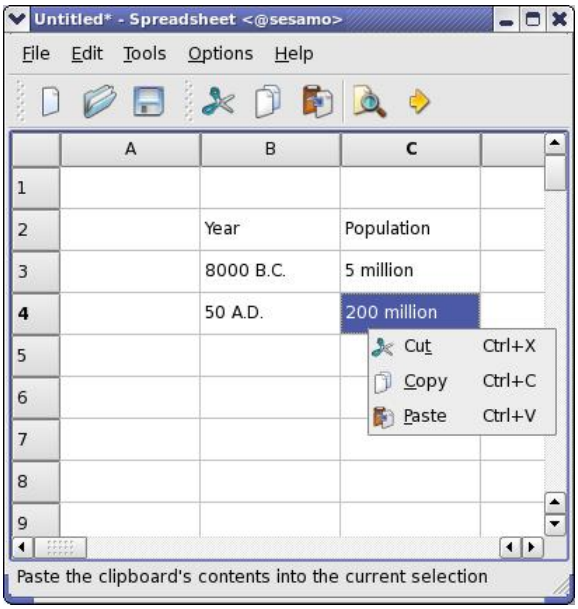
- `QInputDialog`: Útil para introducir una línea de texto o número.

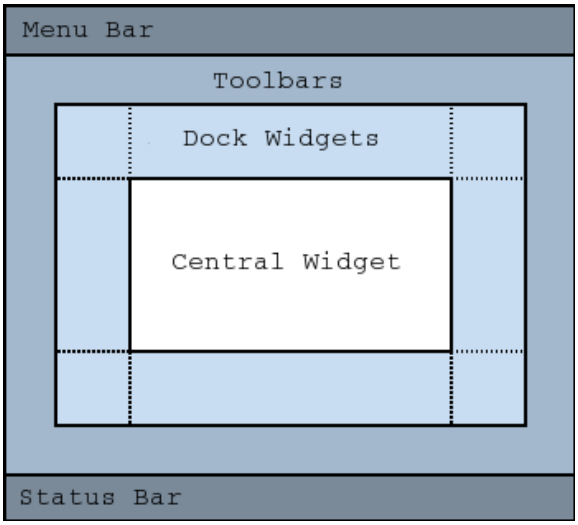
- `QProgressDialog`



5. La ventana principal

- La ventana principal de una aplicación se crea con una subclase de `QMainWindow` (que hereda de `QWidget`).
- Puede crearse también con Qt Designer.
- La ventana puede estar compuesta de:
  - Una **barra de menús**: `QMenuBar`
  - Una o varias **barras de utilidades**: `QToolBar`
    - Están formados por botones que suelen etiquetarse con imágenes.
  - Area principal de la aplicación (***central widget***).
  - **Barra de estado**: `QStatusBar`





5.1. Definición de la clase ventana principal

- Crearemos una clase que herede de `QMainWindow`

Ejemplo: chap03/spreadsheet/mainwindow.h

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H
#include <QMainWindow>
class QAction;
class QLabel;
class FindDialog;
class Spreadsheet;
class MainWindow : public QMainWindow
{
    Q_OBJECT
public:
    MainWindow();
protected:
    void closeEvent(QCloseEvent *event);
private slots:
    void newFile();
    void open();
    bool save();
    bool saveAs();
    void find();
    void goToCell();
    void sort();
    void about();
    void openRecentFile();
    void updateStatusBar();
    void spreadsheetModified();
private:
    void createActions();
    void createMenus();
    void createContextMenu();
    void createToolBars();
    void createStatusBar();
    void readSettings();
    void writeSettings();
    bool okToContinue();
    bool loadFile(const QString &fileName);
    bool saveFile(const QString &fileName);
```

```
    void setCurrentFile(const QString &fileName);
    void updateRecentFileActions();
    QString strippedName(const QString &fullFileName);
    Spreadsheet *spreadsheet;
    FindDialog *findDialog;
    QLabel *locationLabel;
    QLabel *formulaLabel;
    QStringList recentFiles;
    QString curFile;
    enum { MaxRecentFiles = 5 };
    QAction *recentFileActions[MaxRecentFiles];
    QAction *separatorAction;
    QMenu *fileMenu;
    QMenu *editMenu;
    QMenu *selectSubMenu;
    QMenu *toolsMenu;
    QMenu *optionsMenu;
    QMenu *helpMenu;
    QToolBar *fileToolBar;
    QToolBar *editToolBar;
    QAction *newAction;
    QAction *openAction;
    QAction *saveAction;
    QAction *saveAsAction;
    QAction *exitAction;
    QAction *cutAction;
    QAction *copyAction;
    QAction *pasteAction;
    QAction *deleteAction;
    QAction *selectRowAction;
    QAction *selectColumnAction;
    QAction *selectAllAction;
    QAction *findAction;
    QAction *goToCellAction;
    QAction *recalculateAction;
    QAction *sortAction;
    QAction *showGridAction;
    QAction *autoRecalcAction;
    QAction *aboutAction;
    QAction *aboutQtAction;
};
#endif
```

- El widget `spreadsheet` de la clase `Spreadsheet` (subclase de `QTableWidget`) será el widget central de la ventana principal.
- La función `closeEvent()` es una función virtual de `QWidget` que se llama automáticamente cuando el usuario cierra la ventana.

5.2. Constructor de la clase

Ejemplo: chap03/spreadsheet/mainwindow.cpp

```
#include <QtGui>
#include "finddialog.h"
#include "gotocelldialog.h"
#include "mainwindow.h"
#include "sortdialog.h"
#include "spreadsheet.h"
MainWindow::MainWindow()
{
    spreadsheet = new Spreadsheet;
    setCentralWidget(spreadsheet);
    createActions();
    createMenus();
    createContextMenu();
    createToolBars();
    createStatusBar();
    readSettings();
    findDialog = 0;
    setWindowIcon(QIcon(":/images/icon.png"));
    setCurrentFile("");
}
```

5.3. El icono de la aplicación y mecanismo de recursos

- A una ventana principal se le puede asociar un icono con `setWindowIcon`.
- Los mecanismos que permiten usar imágenes en Qt son:
  - Cargar las imágenes en tiempo de ejecución de ficheros.
  - Incluir (con `#include`) en el código fuente ficheros XPM.

  - Usar el mecanimo de recursos.

- En este caso la imagen del icono se ha proporcionado mediante el *mecanismo de recursos* de Qt.
  - Se necesita crear un fichero de recursos (`spreadsheet.qrc`):

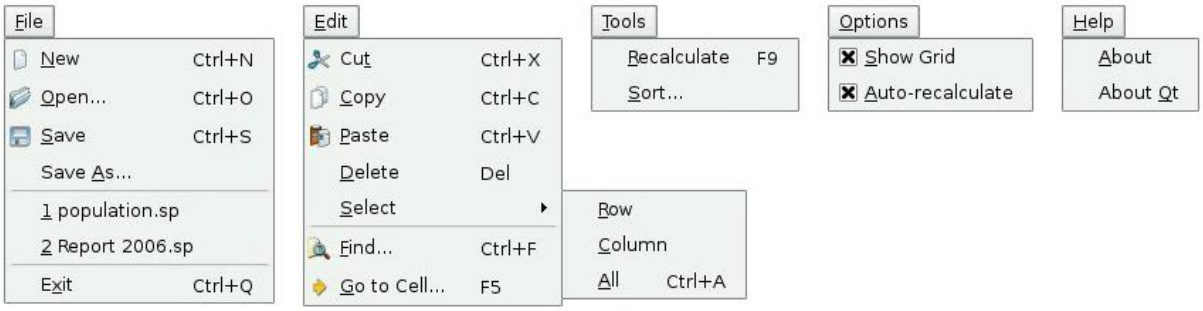
```
<!DOCTYPE RCC><RCC version="1.0">
<qresource>
    <file>images/icon.png</file>
    ...
    <file>images/gotocell.png</file>
</qresource>
</RCC>
```

  - El fichero de recursos se debe incluir en el fichero de proyecto `.pro`:

```
RESOURCES         = spreadsheet.qrc
```

  - En el código fuente nos referiremos a cada recurso anteponiendo `:/`

5.4. Creación de menús y toolbars



- Los menús los crearemos con el *mecanismo de acciones*.
- Una **acción** es un item que puede añadirse a varios menús y toolbars.
- Las etapas para crear los menús y toolbars son:
  - Crear e inicializar las acciones.
  - Crear los menús y colocarle las acciones.
  - Crear los toolbars y colocarle las acciones.

5.4.1. Creación de las acciones

- Las acciones suelen crearse como hijas de la ventana principal
- Cada acción puede tener asociado un texto, un icono, una *tecla aceleradora*, un texto para la barra estado y un *tooltip*.

Ejemplo: chap03/spreadsheet/mainwindow.cpp

```
void MainWindow::createActions()
{
    newAction = new QAction(tr("&New"), this);
    newAction->setIcon(QIcon(":/images/new.png"));
    newAction->setShortcut(tr("Ctrl+N"));
    newAction->setStatusTip(tr("Create a new spreadsheet file"));
    connect(newAction, SIGNAL(triggered()), this, SLOT(newFile()));
    ...
    showGridAction = new QAction(tr("&Show Grid"), this);
    showGridAction->setCheckable(true);
    showGridAction->setChecked(spreadsheet->showGrid());
    showGridAction->setStatusTip(tr("Show or hide the "
                                    "spreadsheet's grid"));
    connect(showGridAction, SIGNAL(toggled(bool)),
            spreadsheet, SLOT(setShowGrid(bool)));
    ...
    aboutQtAction = new QAction(tr("About &Qt"), this);
    aboutQtAction->setStatusTip(tr("Show the Qt library's "
                                   "About box"));
    connect(aboutQtAction, SIGNAL(triggered()),
            qApp, SLOT(aboutQt()));
```

- La clase `QActionGroup` permite definir acciones mutuamente exclusivas.
- El slot `aboutQt()` de `qApp` muestra un diálogo *about* sobre la versión usada de Qt.

5.4.2. Creación de los menús

- El método `QMainWindow::menuBar()` crea el `QMenuBar` (barra de menús) la primera vez que se llama.
- El método `QMenuBar::addMenu()` crea un `QMenu` y lo añade a la barra de menús.

Ejemplo: chap03/spreadsheet/mainwindow.cpp

```
void MainWindow::createMenus()
{
    fileMenu = menuBar()->addMenu(tr("&File"));
    fileMenu->addAction(newAction);
    fileMenu->addAction(openAction);
    fileMenu->addAction(saveAction);
    fileMenu->addAction(saveAsAction);
    separatorAction = fileMenu->addSeparator();
    for (int i = 0; i < MaxRecentFiles; ++i)
        fileMenu->addAction(recentFileActions[i]);
    fileMenu->addSeparator();
    fileMenu->addAction(exitAction);
```

- Los submenús se crean también con `QMenu::addMenu()`

Ejemplo: chap03/spreadsheet/mainwindow.cpp

```
    editMenu = menuBar()->addMenu(tr("&Edit"));
    editMenu->addAction(cutAction);
    editMenu->addAction(copyAction);
    editMenu->addAction(pasteAction);
    editMenu->addAction(deleteAction);

    selectSubMenu = editMenu->addMenu(tr("&Select"));
    selectSubMenu->addAction(selectRowAction);
    selectSubMenu->addAction(selectColumnAction);
    selectSubMenu->addAction(selectAllAction);
```

5.4.3. Creación de menús contextuales

- Cualquier widget puede tener asociado una lista de `QAction`s que pueden usarse por ejemplo para construir un menú contextual.
- En tal caso usaremos la siguiente llamada para indicar que el menú contextual se construye con la lista de acciones:

  `widget->setContextMenuPolicy(Qt::ActionsContextMenu);`

Ejemplo: chap03/spreadsheet/mainwindow.cpp

```
void MainWindow::createContextMenu()
{
    spreadsheet->addAction(cutAction);
    spreadsheet->addAction(copyAction);
    spreadsheet->addAction(pasteAction);
    spreadsheet->setContextMenuPolicy(Qt::ActionsContextMenu);
}
```

5.4.4. Creación de barras de utilidades

- Un *toolbar* se crea con `QMainWindow::addToolBar()`
- Los elementos se añaden entonces con `QToolBar::addAction()`

Ejemplo: chap03/spreadsheet/mainwindow.cpp

```
void MainWindow::createToolBars()
{
    fileToolBar = addToolBar(tr("&File"));
    fileToolBar->addAction(newAction);
    fileToolBar->addAction(openAction);
    fileToolBar->addAction(saveAction);
    editToolBar = addToolBar(tr("&Edit"));
    editToolBar->addAction(cutAction);
    editToolBar->addAction(copyAction);
    editToolBar->addAction(pasteAction);
    editToolBar->addSeparator();
    editToolBar->addAction(findAction);
    editToolBar->addAction(goToCellAction);
}
```

5.5. Creación de la barra de estado

- La función `QMainWindow::statusBar()` crea la barra de estado la primera vez que se llama.
- A la barra de estado se añadirán `QLabel`s con `QStatusBar::addWidget()`

Ejemplo: chap03/spreadsheet/mainwindow.cpp

```
void MainWindow::createStatusBar()
{
    locationLabel = new QLabel(" W999 ");
    locationLabel->setAlignment(Qt::AlignHCenter);
    locationLabel->setMinimumSize(locationLabel->sizeHint());
    formulaLabel = new QLabel;
    formulaLabel->setIndent(3);
    statusBar()->addWidget(locationLabel);
    statusBar()->addWidget(formulaLabel, 1);
    connect(spreadsheet, SIGNAL(
            currentCellChanged(int, int, int, int)),
            this, SLOT(updateStatusBar()));
    connect(spreadsheet, SIGNAL(modified()),
            this, SLOT(spreadsheetModified()));
    updateStatusBar();
}

void MainWindow::updateStatusBar()
{
    locationLabel->setText(spreadsheet->currentLocation());
    formulaLabel->setText(spreadsheet->currentFormula());
}

void MainWindow::spreadsheetModified()
{
    setWindowModified(true);
    updateStatusBar();
}
```

- Puede mostrarse un mensaje durante un tiempo determinado con: `statusBar()->showMessage(tr("Mensaje"), 2000);`

5.6. Implementación de los slots

Ejemplo: chap03/spreadsheet/mainwindow.cpp

```
void MainWindow::newFile()
{
    if (okToContinue()) {
        spreadsheet->clear();
        setCurrentFile("");
    }
}

bool MainWindow::okToContinue()
{
    if (isWindowModified()) {
        int r = QMessageBox::warning(this, tr("Spreadsheet"),
                        tr("The document has been modified.\n"
                           "Do you want to save your changes?"),
                        QMessageBox::Yes | QMessageBox::Default,
                        QMessageBox::No,
                        QMessageBox::Cancel | QMessageBox::Escape);
        if (r == QMessageBox::Yes) {
            return save();
        } else if (r == QMessageBox::Cancel) {
            return false;
        }
    }
    return true;
}
```

- Cuando se asigna un slot a varias acciones probablemente necesitemos usar `QObject::sender()`:

Ejemplo: chap03/spreadsheet/mainwindow.cpp

```
void MainWindow::openRecentFile()
{
    if (okToContinue()) {
        QAction *action = qobject_cast<QAction *>(sender());
        if (action)
            loadFile(action->data().toString());
    }
}
```

5.7. Introducción al manejo de eventos de bajo nivel

- Eventos tales como pulsación de botones de ratón, movimiento de ratón se conocen como *eventos de bajo nivel* o *sintácticos*.
- En Qt se controlan creando subclases de algún Widget Qt, y sobreescribiendo el *método virtual* correspondiente de `QWidget`:
  - `mousePressEvent(QMouseEvent*)`
  - `mouseMoveEvent(QMouseEvent*)`
  - `paintEvent(QPaintEvent*)`
  - `resizeEvent(QResizeEvent*)`
  - `closeEvent(QCloseEvent *)`
  - `keyPressEvent(QKeyEvent * event )`
- Cuando el usuario cierra la ventana se llama automáticamente a la función `closeEvent(QCloseEvent *)`, que podemos sobreescribir para adaptarla a nuestras necesidades:

Ejemplo: chap03/spreadsheet/mainwindow.cpp

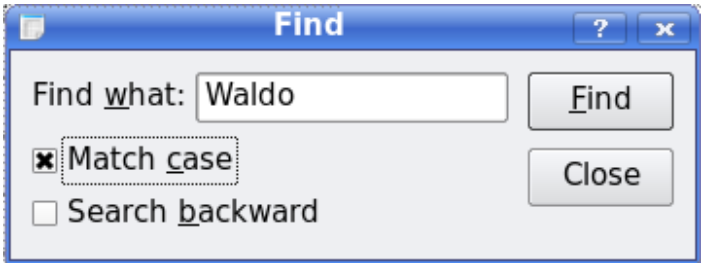
```
void MainWindow::closeEvent(QCloseEvent *event)
{
    if (okToContinue()) {
        writeSettings();
        event->accept();
    } else {
        event->ignore();
    }
}
```

5.8. Uso de nuestros diálogos

- Los diálogos que hemos implementado pueden usarse ahora desde la ventana principal:

Ejemplo con diálogo no modal:
chap03/spreadsheet/mainwindow.cpp

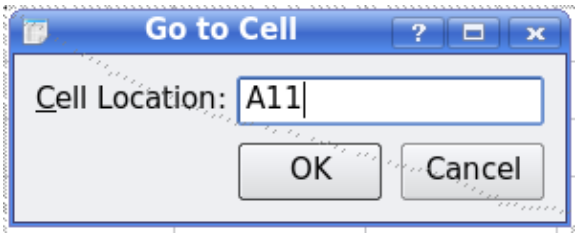
```
void MainWindow::find()
{
    if (!findDialog) {
        findDialog = new FindDialog(this);
        connect(findDialog, SIGNAL(findNext(const QString &,
                                           Qt::CaseSensitivity)),
                spreadsheet, SLOT(findNext(const QString &,
                                           Qt::CaseSensitivity)));
        connect(findDialog, SIGNAL(findPrevious(const QString &,
                                           Qt::CaseSensitivity)),
                spreadsheet, SLOT(findPrevious(const QString &,
                                           Qt::CaseSensitivity)));
    }
    findDialog->show();
    findDialog->activateWindow();
}
```



- Los diálogos modales suelen crearse en la *pila* en lugar de con memoría dinámica, ya que no se necesitan una vez que se usan:

Ejemplo con diálogo modal: chap03/spreadsheet/mainwindow.cpp

```
void MainWindow::goToCell()
{
    GoToCellDialog dialog(this);
    if (dialog.exec()) {
        QString str = dialog.lineEdit->text().toUpper();
        spreadsheet->setCurrentCell(str.mid(1).toInt() - 1,
                                    str[0].unicode() - 'A');
    }
}
```



- Podemos asociar nuestro propio diálogo *about* usando la clase `QMessageBox`:

Ejemplo con diálogo modal: chap03/spreadsheet/mainwindow.cpp

```
void MainWindow::about()
{
    QMessageBox::about(this, tr("About Spreadsheet"),
            tr("<h2>Spreadsheet 1.1</h2>"
               "<p>Copyright &copy; 2006 Software Inc."
               "<p>Spreadsheet is a small application that "
               "demonstrates QAction, QMainWindow, QMenuBar, "
               "QStatusBar, QTableWidget, QToolBar, and many other "
               "Qt classes."));
}
```

5.9. Almacenamiento de la configuración de la aplicación

- Qt permite almacenar de forma *persistente* las propiedades de la configuración actual de una aplicación mediante la clase `QSettings`: tamaño de ventana, posición, opciones, últimos ficheros abiertos, etc
- Las propiedades se almacenan en un fichero que depende de la plataforma:
  - En Windows se usa el registro del sistema.
  - En Unix se usa ficheros de texto. En mi distribución se colocan en el directorio `$HOME/.config`
- Para usarlo tendremos que crear un objeto de `QSettings` de la siguiente forma:

  ```
  QSettings settings("Micompañia", "Nombre de mi aplicación");
  ```

- `QSettings` almacena cada propiedad mediante un nombre de la propiedad (`QString`) y un valor (`QVariant`)
- Para escribir un valor usaremos `setValue(QString,QVariant)`

  ```
  settings.setValue("editor/Margen", 68);
  ```

- Para leer el valor usaremos `value(QString propiedad)` o `value(QString propiedad,QVariant valorpordefecto)`:

  ```
  int margin = settings.value("editor/Margen").toInt();
  ```

Ejemplo: chap03/spreadsheet/mainwindow.cpp

```
void MainWindow::writeSettings()
{
    QSettings settings("Software Inc.", "Spreadsheet");

    settings.setValue("geometry", geometry());
    settings.setValue("recentFiles", recentFiles);
    settings.setValue("showGrid", showGridAction->isChecked());
    settings.setValue("autoRecalc", autoRecalcAction->isChecked());
}
```

Ejemplo: chap03/spreadsheet/mainwindow.cpp

```
void MainWindow::readSettings()
{
    QSettings settings("Software Inc.", "Spreadsheet");
    QRect rect = settings.value("geometry",
                                QRect(200, 200, 400, 400)).toRect();
    move(rect.topLeft());
    resize(rect.size());
    recentFiles = settings.value("recentFiles").toStringList();
    updateRecentFileActions();
    bool showGrid = settings.value("showGrid", true).toBool();
    showGridAction->setChecked(showGrid);
    bool autoRecalc = settings.value("autoRecalc", true).toBool();
    autoRecalcAction->setChecked(autoRecalc);
}
```

5.10. Implementación del widget central

- El área central de una ventana principal puede ocuparse con cualquier widget. Por ejemplo:
  - Con un widget Qt estándar: `QTableWidget`, `QTextEdit`, etc
  - Con un widget optimizado: una subclase de algún tipo de widget.
  - Con un widget `QWidget` y un manejador de posicionamiento para incluir otros widgets.
  - Con un widget `QWorkspace` (para aplicaciones MDI).

Ejemplo: chap03/spreadsheet/mainwindow.cpp

```
MainWindow::MainWindow()
{
    spreadsheet = new Spreadsheet;
    setCentralWidget(spreadsheet);
    ...
}
```

6. Sistema de dibujo de Qt

- El sistema de dibujo de Qt está basado en las clases `QPainter`, `QPaintDevice` y `QPaintEngine`:
  - `QPainter` se usa para llamar a las primitivas de dibujo: puntos, líneas, rectángulos, elipses, arcos, polígonos, curvas de Bézier, pixmaps, imágenes, texto, etc.
  - `QPaintDevice` es la abstracción de un espacio bidimensional donde podemos dibujar usando un objeto `QPainter`.

    `QPainter` puede actuar sobre cualquier objeto que herede de la clase `QPaintDevice`.
  - `QPaintEngine` proporciona el interfaz que usa el objeto `QPainter` para dibujar en distintos dispositivos.

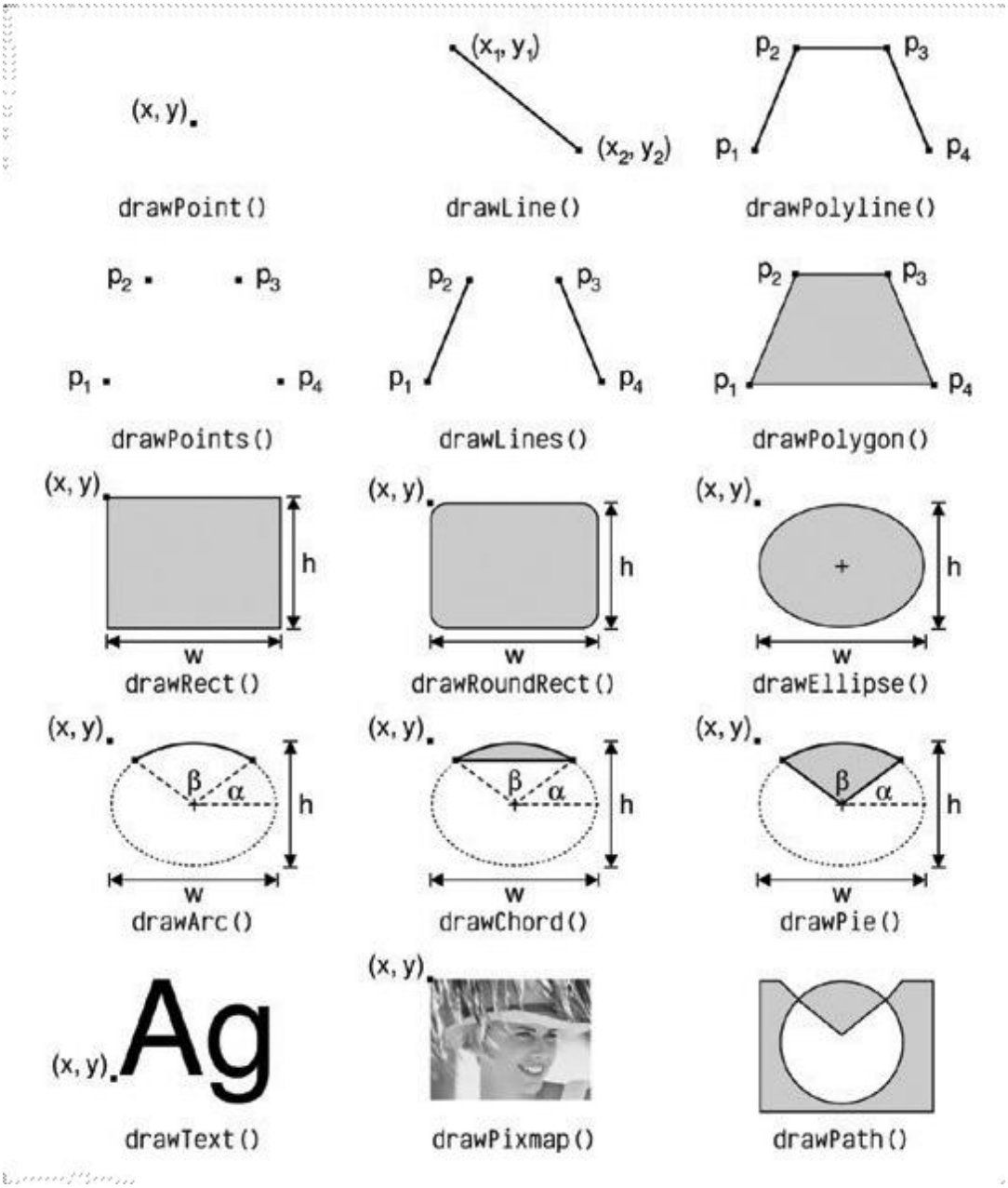
    Esta clase es usada internamente por Qt, y los programadores no la usarán salvo que quieran programar nuevos dispositivos.
- Puede usarse tanto para dibujar en dispositivos de dibujo (`QWidget`, `QPixmap`, o `QImage`) como en dispositivos de impresión (junto con la clase `QPrinter`) o para generar ficheros pdf.

6.1. Dibujar con QPainter

- Para comenzar a dibujar en un dispositivo de dibujo (por ejemplo un widget) creamos un objeto `QPainter` pasándole al constructor un puntero al dispositivo. Por ejemplo:

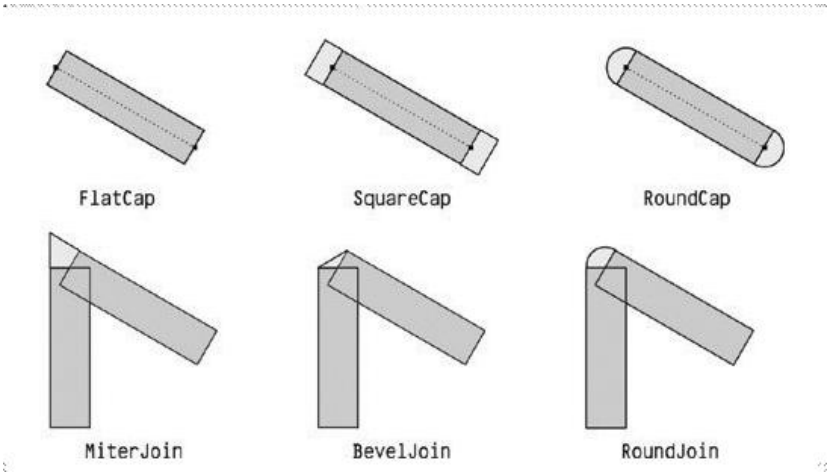
```
void MyWidget::paintEvent(QPaintEvent *event)
{
    QPainter painter(this);
    ...
}
```

- Con el objeto `QPainter` podemos dibujar varias formas:

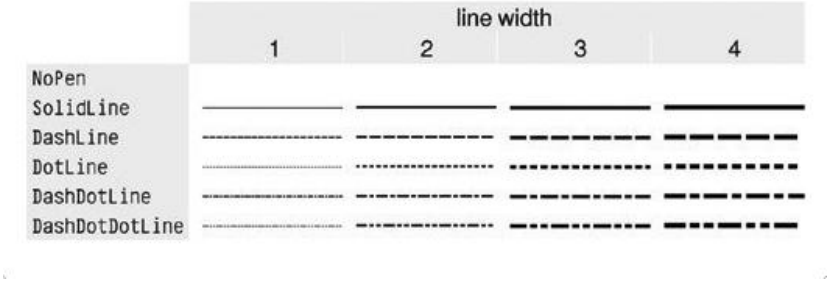


- El resultado de cada primitiva gráfica depende de los atributos del `QPainter`. Los más importantes son:
  - **Pincel** (*pen*): Se usa para dibujar líneas y bordes de formas (rectángulos, elipses, etc). Consiste de un color (`QColor`), una anchura, estilo de línea, *cap style* y *join style*.
  - **Brocha** (*brush*): Patrón usado para rellenar formas geométricas. Consiste normalmente de un color y un estilo, pero puede ser también una textura (pixmap que se repite infinitamente) o un gradiente.
  - **Font**: Se usa al dibujar texto. Contiene muchos atributos tal como la familia y el tamaño de punto.
- Tales atributos pueden modificarse en cualquier momento con `setPen()`, `setBrush` y `setFont()`.

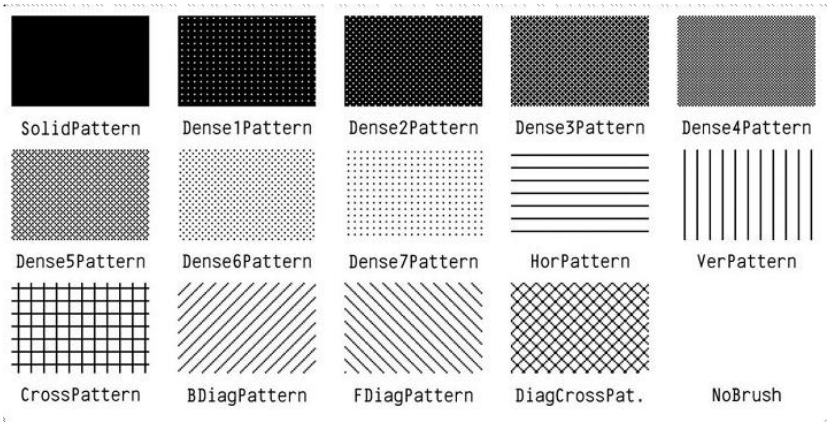
Estilos Cap y join



Estilos de pincel



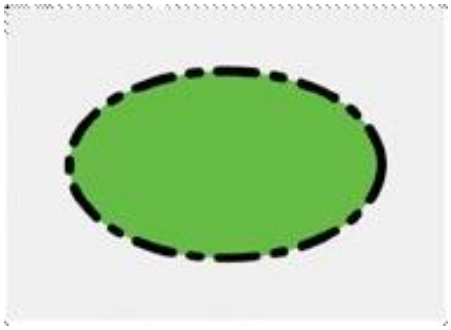
Estilos de brocha



Ejemplo 1:

```
QPainter painter(this);
painter.setRenderHint(QPainter::Antialiasing, true);
painter.setPen(QPen(Qt::black, 12, Qt::DashDotLine, Qt::RoundCap));
painter.setBrush(QBrush(Qt::green, Qt::SolidPattern));
painter.drawEllipse(80, 80, 400, 240);
```

- La llamada a `setRenderHint()` activa el ***antialiasing***, que hace que se usen diferentes intesidades de color en las fronteras para reducir la distorsión visual al convertir las fronteras de una figura en pixels.



Ver ejemplo de Antialiasing en: Aplicación `concentriccircles`.

Ejemplo 2:

```
QPainter painter(this);
painter.setRenderHint(QPainter::Antialiasing, true);
painter.setPen(QPen(Qt::black, 15, Qt::SolidLine, Qt::RoundCap,
                    Qt::MiterJoin));
painter.setBrush(QBrush(Qt::blue, Qt::DiagCrossPattern));
painter.drawPie(80, 80, 400, 240, 60 * 16, 270 * 16);
```

Ejemplo 3:

```
QPainter painter(this);
painter.setRenderHint(QPainter::Antialiasing, true);
QPainterPath path;
path.moveTo(80, 320);
path.cubicTo(200, 80, 320, 80, 480, 320);
painter.setPen(QPen(Qt::black, 8));
painter.drawPath(path);
```

- La clase `QPainterPath` permite especificar un dibujo vectorial mediante la unión de varios elementos gráficos básicos: líneas rectas, elipses, polígonos, arcos, curvas cuadráticas o de Bezier, y otros objetos `QPainterPath`.
- El objeto `QPainterPath` especifica el borde de una figura, que puede rellenarse si usamos una brocha en el `QPainter`

6.2. Otros atributos del QPainter

- **Brocha del background**: Es la brocha usada al dibujar texto opaco, líneas tipo *stippled* y bitmaps. Esta brocha no tiene efecto en modo de background transparente (el de por defecto).
  - `setBackgroundBrush(QBrush)`
  - `setBackgroundMode( BGMode)`: Establece el modo de background (`Qt::TransparentMode`, `Qt::OpaqueMode`)
- **Origen de la brocha**: Punto inicial para patrones de rellenado con la brocha (esquina superior izquierda normalmente).
- **Máscara de recorte** (*clip region*): Área del dispositivo que se afectará por las primitivas gráficas.
- **Viewport**, **window** y **world matrix**: Determinan cómo transformar las coordenadas del `QPainter` en coordenadas físicas del dispositivo.
- **Modo de composición**: Especifica cómo combinar los píxeles existentes con los que se van a dibujar.
  - `setCompositionMode(CompositionMode)`

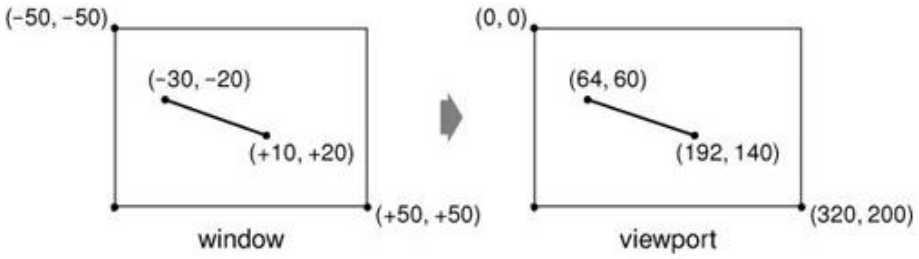
6.3. Transformaciones

- El sistema de coordenadas por defecto de `QPainter` tiene su origen $(0, 0)$ en la esquina superior izquierda.
- Las coordenadas x se incrementan hacia la derecha y las y hacia abajo. Cada pixel ocupa un área de tamaño 1×1.

- Es posible modificar el sistema de coordenadas por defecto usando el *viewport*, *window* y *world matrix*
  - El **viewport** es un rectángulo que especifica las *coordenadas físicas*.
  - La **window** especifica el mismo rectángulo pero en *coordenadas lógicas*.
  - Cuando dibujamos con una primitiva gráfica, se usan coordenadas lógicas, que se trasladan en coordenadas físicas usando el **viewport** y **window** actuales.
  - Por defecto, ambos rectángulos coinciden con el rectángulo del dispositivo.
    Ejemplo: Si el dispositivo es un widget de 320×200, el viewport y window son también un rectángulo de 320×200 con su esquina superior izquierda en la posición $(0, 0)$.
    En este caso, el sistema de coordenadas físico y lógico son el mismo.
- Este sistema hace que el código para dibujar pueda hacerse independiente del tamaño o resolución del dispositivo.
  Ejemplo: En el caso de antes, podríamos definir el sistema de coordenadas lógicas en el rango $(-50, -50)$ a $(50, 50)$, siendo $(0, 0)$ el centro mediante:

  ```
  painter.setWindow(-50, -50, 100, 100);
  ```

  - Ahora la coordenada lógica $(-50, -50)$ corresponde con la coordenada física $(0, 0)$
  - La coordenada lógica $(50, 50)$ corresponde con la coordenada física $(320, 200)$



- La **world matrix** es una matriz de trasformación que es aplicada adicionalmente además del viewport y window.
  - Permite hacer las siguientes trasformaciones a los items que dibujamos: trasladar, escalar, rotar y girar (*shear*). (Ver programa de ejemplo `affine` en `/usr/lib/qt4/demos/affine`)

    Ejemplo: Para dibujar un texto rotado 45º usaremos

    ```
    QMatrix matrix;
    matrix.rotate(45.0);
    painter.setMatrix(matrix);
    painter.drawText(rect, Qt::AlignCenter, tr("Revenue"));
    ```

  - Si especificamos varias trasformaciones, se aplicarán en el orden que demos.

    Ejemplo: Si queremos usar el punto $(10, 20)$ como punto de rotación, podemos trasladar primero la window, hacer la rotación y trasladar la window de nuevo a su posición original.

    ```
    QMatrix matrix;
    matrix.translate(-10.0, -20.0);
    matrix.rotate(45.0);
    matrix.translate(+10.0, +20.0);
    painter.setMatrix(matrix);
    painter.drawText(rect, Qt::AlignCenter, tr("Revenue"));
    ```

  - Una forma más simple de especificar trasformaciones es usando los métodos `translate()`, `scale()`, `rotate()`, y `shear()` de `QPainter`.

    Ejemplo:

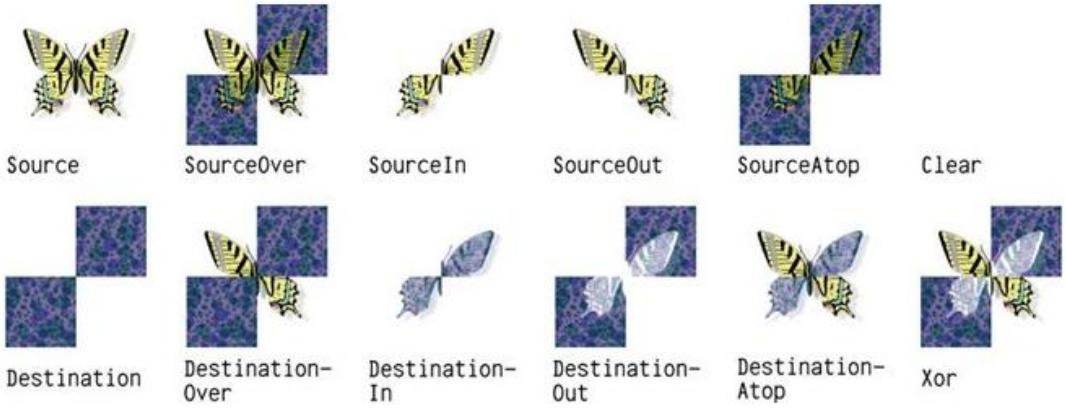
    ```
    painter.translate(-10.0, -20.0);
    painter.rotate(45.0);
    painter.translate(+10.0, +20.0);
    painter.drawText(rect, Qt::AlignCenter, tr("Revenue"));
    ```

  - Pero si queremos usar varias veces las mismas trasformaciones, es más eficiente almacenarlas en un objeto `QMatrix` y definir la *window matrix* del objeto `QPainter` cuando se necesiten tales trasformaciones.

6.4. Modos de composición

- Otra ventaja del motor de dibujo de Qt es que soporta *modos de composición*: forma en que se combian el fuente y destino.

  `QPainter::setCompositionMode()`



Ejemplo de uso de un modo de composición:

```
QImage resultImage = checkerPatternImage;
QPainter painter(&resultImage);
painter.setCompositionMode(QPainter::CompositionMode_Xor);
painter.drawImage(0, 0, butterflyImage);
```

6.5. Rendering de alta calidad con QImage

- En X11 y MAC OS X, el peso de dibujar en un `QWidget` o `QPixmap` recae en el motor de dibujo de la plataforma nativa.
- En X11 esto hace que la comunicación cliente-servidor se minimice, ya que los Pixmaps se almacenan en el servidor.
- Pero tiene el inconveniente que se puede perder calidad en algunos gráficos en plataformas X11 que no tengan instalada la *X Render Extension*:
  - **Por ejemplo**: No se puede usar *Antialiasing*.
- Si la calidad es más importante que la eficiencia podemos dibujar en un `QImage` y copiar luego el resultado en pantalla
  - Así, se usará el motor interno de Qt para dibujar.
  - Esto permite obtener el mismo resultado en cualquier plataforma.
  - Para ello, el objeto `QImage` debe crearse de tipo `QImage::Format_RGB32` o `QImage::Format_ARGB32_Premultiplied`.

Ejemplo de uso de Antialiasing sin QImage:

```
void MyWidget::paintEvent(QPaintEvent *event)
{
    QPainter painter(this);
    painter.setRenderHint(QPainter::Antialiasing, true);
    draw(&painter);
}
```

Ejemplo de uso de Antialiasing con QImage:

```
void MyWidget::paintEvent(QPaintEvent *event)
{
    QImage image(size(), QImage::Format_ARGB32_Premultiplied);
    QPainter imagePainter(&image);
    imagePainter.initFrom(this);
    imagePainter.setRenderHint(QPainter::Antialiasing, true);
    imagePainter.eraseRect(rect());
    draw(&imagePainter);
    imagePainter.end();
    QPainter widgetPainter(this);
    widgetPainter.drawImage(0, 0, image);
}
```

6.6. Clases QImage y QPixmap

- Qt proporciona cuatro clases para manejar imágenes: QImage, QPixmap, QBitmap y QPicture.
  - `QImage`: Está diseñada y optimizada para operaciones de entrada/salida, y para acceso y manipulación directa a nivel de pixel.
  - `QPixmap`: Está diseñada y optimizada para mostrar imágenes en pantalla.
  - `QBitmap`: Es una clase que hereda de `QPixmap`, para pixmaps de profundidad 1.
  - `QPicture`: Es un dispositivo de dibujo que permite almacenar y reproducir comandos `QPainter`.
- Todos heredan de `QPaintDevide` por lo que `QPainter` podrá usarse directamente para dibujar sobre ellos.

6.6.1. Clase QImage

- Una imagen puede ser **cargada** de un fichero con el constructor o bien con `QImage::load()`
- El fichero podría ser un fichero normal (cargado en tiempo de ejecución) o uno leido con el *sistema de recursos* (en tiempo de compilación).
- Una imagen puede ser grabada a disco con `save()`.
- Por defecto Qt soporta los siguientes formatos:

Format	Description	Qt's support
BMP	Windows Bitmap	Read/write
GIF	Graphic Interchange Format (optional)	Read
JPG	Joint Photographic Experts Group	Read/write
JPEG	Joint Photographic Experts Group	Read/write
PNG	Portable Network Graphics	Read/write
PBM	Portable Bitmap	Read
PGM	Portable Graymap	Read
PPM	Portable Pixmap	Read/write
TIFF	Tagged Image File Format	Read/write
XBM	X11 Bitmap	Read/write
XPM	X11 Pixmap	Read/write

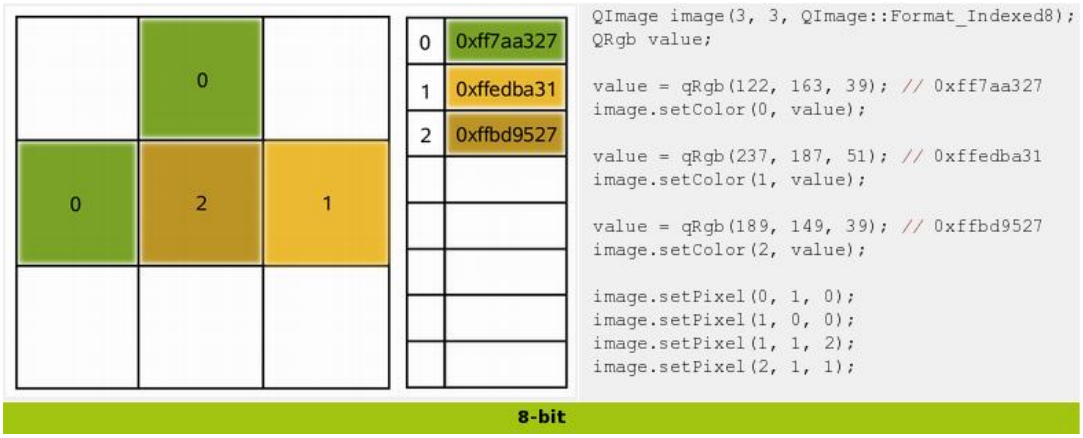
- Las funciones para manipular una imagen dependen del *formato* (`QImage::Format`) usado al crear el objeto `QImage`:

Constant	Value	Description
QImage::Format_Invalid	0	The image is invalid.
QImage::Format_Mono	1	The image is stored using 1-bit per pixel. Bytes are packed with the most significant bit (MSB) first.
QImage::Format_MonoLSB	2	The image is stored using 1-bit per pixel. Bytes are packed with the less significant bit (LSB) first.
QImage::Format_Indexed8	3	The image is stored using 8-bit indexes into a colormap.
QImage::Format_RGB32	4	The image is stored using a 32-bit RGB format (0xffRRGGBB).
QImage::Format_ARGB32	5	The image is stored using a 32-bit ARGB format (0xAARRGGBB).
QImage::Format_ARGB32_Premultiplied	6	The image is stored using a premultiplied 32-bit ARGB format (0xAARRGGBB), i.e. the red, green, and blue channels are multiplied by the alpha component divided by 255. (If RR, GG, or BB has a higher value than the alpha channel, the results are undefined.) Certain operations (such as image composition using alpha blending) are faster using premultiplied ARGB32 than with plain ARGB32.

- **Imágenes de 32 bits** usan valores ARGB.

  - Cada valor de pixel (32 bits) se descompone en 8 bits para la intensidad de rojo, 8 para verde y 8 para azul, y 8 para nivel de trasparencia.(el componente *alpha* u *opacidad*).
    Por ejemplo: El color rojo puro se representaría con:
    `QRgb red = qRgba(255, 0, 0, 255);`
    o bien con:
    `QRgb red = qRgb(255, 0, 0);` o bien con:
    `QRgb red = 0xFFFF0000;`

- **Imágenes monocromo** y de 8 bits usan valores basados en índices en la paleta de color.



  - Ahora el valor del pixel es un índice en la tabla (paleta) de color de la imagen.

6.6.2. Clase QPixmap

- Una imagen puede ser **cargada** de un fichero con el constructor o bien con `QPixmap::load()`
- Un pixmap puede crearse con uno de los constructores o con las funciones estáticas:
  - `grabWidget()`: Crea un pixmap con el contenido capturado de un widget.
  - `grabWindow()`: Crea un pixmap con el contenido capturado de una ventana.
- Un `QPixmap` puede mostrarse en pantalla con un `QLabel` o alguna de las subclases de `QAbstractButton` (como `QPushButton` y `QToolButton`):
  - `QLabel` tiene la propiedad `pixmap` y las funciones de acceso `pixmap()` y `setPixmap()`.
  - `QAbstractButton` tiene la propiedad `icon` (`QIcon`) y las funciones de acceso `icon()` y `setIcon()`.
- Los datos de cada pixel sólo pueden ser accedidos a través de funciones de la clase `QPainter` o convirtiendo el `QPixmap` en un `QImage`:

  En un pixmap los datos de cada pixel son datos internos manejados por el correspondiente manejador de ventanas (o servidor X).
- Un `QPixmap` puede convertirse en `QImage` con `QPixmap::toImage()`
- Un `QImage` puede convertirse en `QPixmap` con `QPixmap::fromImage()`

6.6.3. Clase QPicture

Ejemplo de grabación de un QPicture en un fichero:

```
QPicture picture;
QPainter painter;
painter.begin(&picture);           // paint in picture
painter.drawEllipse(10,20, 80,70); // draw an ellipse
painter.end();                     // painting done
picture.save("drawing.pic");       // save picture
```

Ejemplo de dibujo de un QPicture en un widget:

```
QPicture picture;
picture.load("drawing.pic");           // load picture
QPainter painter;
painter.begin(&myWidget);              // paint in myWidget
painter.drawPicture(0, 0, picture);    // draw the picture at (0,0)
painter.end();                         // painting done
```

7. Creación de widgets optimizados

- Es posible crear un widget optimizado con una subclase de algún widget Qt o directamente de `QWidget`.
- Esto es preciso cuando necesitamos más de lo que es posible modificando las propiedades de un widget Qt o llamando a las funciones que tenga disponibles.

7.1. Ejemplo: Un spin box hexadecimal

Un spin box hexadecimal: chap05/hexspinbox/hexspinbox.h

```
#ifndef HEXSPINBOX_H
#define HEXSPINBOX_H
#include <QSpinBox>
class QRegExpValidator;
class HexSpinBox : public QSpinBox
{
    Q_OBJECT
public:
    HexSpinBox(QWidget *parent = 0);
protected:
    QValidator::State validate(QString &text, int &pos) const;
    int valueFromText(const QString &text) const;
    QString textFromValue(int value) const;
private:
    QRegExpValidator *validator;
};
#endif
```

- `HexSpinBox` hereda la mayoría de su funcionalidad de `QSpinBox`.
- Añade un constructor típico y sobreescribe tres funciones virtuales de `QSpinBox`.

Un spin box hexadecimal: chap05/hexspinbox/hexspinbox.cpp

```
#include <QtGui>
#include "hexspinbox.h"
HexSpinBox::HexSpinBox(QWidget *parent)
    : QSpinBox(parent)
{
    setRange(0, 255);
    validator = new QRegExpValidator(QRegExp("[0-9A-Fa-f]{1,8}"), this);
}
```

- Se pone como rango por defecto el intervalo $0 - 255$ (`0x00` a `0xFF`) en lugar del que tiene por defecto un `QSpinBox` (0 a 99).
- La variable privada `validator` permitirá validar si las entradas en el editor de líneas del spin box, son válidas:

  Usaremos un `QRegExpValidator` que acepta entre 1 y 8 caracteres, cada uno del conjunto `'0'` a `'9'`, `'A'` a `'F'` y `'a'` a `'f'`.

Un spin box hexadecimal: chap05/hexspinbox/hexspinbox.cpp

```
QValidator::State HexSpinBox::validate(QString &text, int &pos) const
{
    return validator->validate(text, pos);
}
```

- La función `QSpinBox::validate()` es llamada por el `QSpinBox` para ver si el texto introducido es válido.
- `QSpinBox::validate()` puede devolver `Invalid`, `Intermediate` y `Acceptable`.
- La función `QRegExpValidator::validate()` nos permite devolver el resultado deseado.

Un spin box hexadecimal: chap05/hexspinbox/hexspinbox.cpp

```
QString HexSpinBox::textFromValue(int value) const
{
    return QString::number(value, 16).toUpper();
}
```

- La función `QSpinBox::textFromValue()` convierte un valor entero en un string.
- `QSpinBox` la llama para actualizar el texto del editor de líneas, cuando el usuario pulsa las flecha ascendente o descendente.

Un spin box hexadecimal: chap05/hexspinbox/hexspinbox.cpp

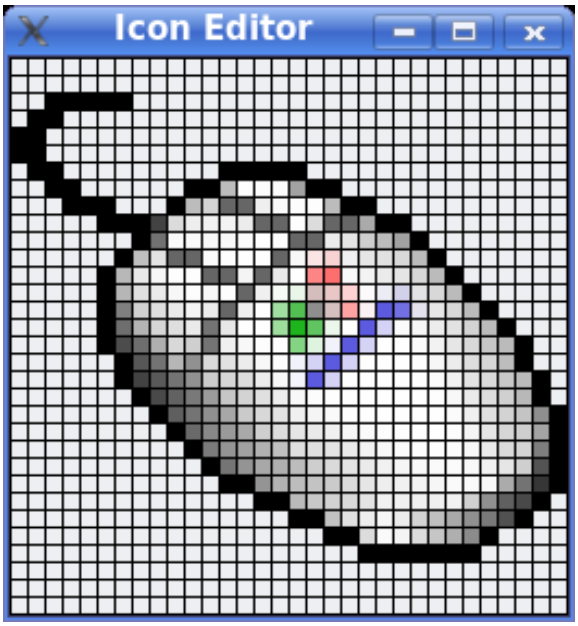
```
int HexSpinBox::valueFromText(const QString &text) const
{
    bool ok;
    return text.toInt(&ok, 16);
}
```

- La función `valueFromText()` realiza la conversión inversa, de string a valor entero.
- `QSpinBox` la llama cuando el usuario introduce un valor en el editor de líneas y pulsa Enter.

7.2. Subclases de QWidget

- Podemos crear widget optimizados combinando los widgets disponibles (widgets Qt u otros widgets optimizados).
- Si el widget que necesitamos no necesita definir sus propias señales y slots, y no sobreescribirá ninguna función virtual, es posible que no haga falta crear una nueva clase, y baste con combinar los widgets disponibles.
- En caso contrario crearemos una nueva clase que heredará de `QWidget`.
  - En ella sobreescribiremos algunos manejadores de eventos de bajo nivel: `paintEvent()`, `mousePressEvent()`, etc.
  - Esto nos permitirá controlar la apariencia y el comportamiento del widget.

7.3. Ejemplo: Un editor de iconos



7.3.1. El fichero .h

Un editor de iconos: chap05/iconeditor/iconeditor.h

```
#ifndef ICONEDITOR_H
#define ICONEDITOR_H
#include <QColor>
#include <QImage>
#include <QWidget>
class IconEditor : public QWidget
{
    Q_OBJECT
    Q_PROPERTY(QColor penColor READ penColor WRITE setPenColor)
    Q_PROPERTY(QImage iconImage READ iconImage WRITE setIconImage)
    Q_PROPERTY(int zoomFactor READ zoomFactor WRITE setZoomFactor)
public:
    IconEditor(QWidget *parent = 0);
    void setPenColor(const QColor &newColor);
    QColor penColor() const { return curColor; }
    void setZoomFactor(int newZoom);
    int zoomFactor() const { return zoom; }
    void setIconImage(const QImage &newImage);
    QImage iconImage() const { return image; }
    QSize sizeHint() const;
```

- `Q_PROPERTY()` es una macro que permite declarar una *propiedad* (campo especial de la clase que permite acceder a ella desde Qt Designer).
- Cada propiedad tiene un tipo (cualquiera soportado por `QVariant`), una función de lectura, y opcionalmente una función de escritura.
- La macro `Q_OBJECT` debe incluirse en la clase cuando definimos propiedades.

Un editor de iconos: chap05/iconeditor/iconeditor.h

```
protected:
    void mousePressEvent(QMouseEvent *event);
    void mouseMoveEvent(QMouseEvent *event);
    void paintEvent(QPaintEvent *event);
private:
    void setImagePixel(const QPoint &pos, bool opaque);
    QRect pixelRect(int i, int j) const;
    QColor curColor;
    QImage image;
    int zoom;
};
#endif
```

- `IconEditor` sobreescribe tres funciones virtuales *protected* de `QWidget`: `mousePressEvent()`, `mouseMoveEvent()` y `paintEvent()`.
- Además define dos funciones y tres variables privadas.

7.3.2. El fichero .cpp

Un editor de iconos: chap05/iconeditor/iconeditor.cpp

```
#include <QtGui>
#include "iconeditor.h"
IconEditor::IconEditor(QWidget *parent)
    : QWidget(parent)
{
    setAttribute(Qt::WA_StaticContents);
    setSizePolicy(QSizePolicy::Minimum, QSizePolicy::Minimum);
    curColor = Qt::black;
    zoom = 8;
    image = QImage(16, 16, QImage::Format_ARGB32);
    image.fill(qRgba(0, 0, 0, 0));
}
```

- Las variables que se inicializan en el constructor son:
  - `curColor` (color del pincel): asignada con color **negro**.
  - `zoom` (factor de zoom): inicializada a valor 8.
  - `image` (imagen del editor de iconos): inicializada con un `QImagen` de tamaño 16×16 y formato ARGB (formato que soporta semi-trasparencia) de 32 bis de profundidad.
- La imagen es rellenada con un color *blanco trasparente* con `image.fill(qRgba(0, 0, 0, 0))`
  `qRgba()` devuelve el color con el tipo `QRgb`.

Un editor de iconos: chap05/iconeditor/iconeditor.cpp

```
QSize IconEditor::sizeHint() const
{
    QSize size = zoom * image.size();
    if (zoom >= 3)
        size += QSize(1, 1);
    return size;
}
```

- `sizeHint()` es una función sobreescrita de `QWidget` que devuelve el tamaño ideal de un widget.
  - En este caso, devolverá el tamaño de la imagen multiplicado por el factor de zoom, y sumándole 1 si el factor de zoom es mayor a 2.
  - El *size hint* de un widget es sobre todo tenido en cuenta, cuando se coloca el widget con un *layout*. Un *layout manager* intenta respectarlo lo máximo posible.
- La llamada a `setSizePolicy(QSizePolicy::Minimum, QSizePolicy::Minimum);` en el constructor informa a cualquier *layout manager* que contenga este widget que el tamaño especificado con `sizeHint()` es realmente un tamaño mínimo.

Un editor de iconos: chap05/iconeditor/iconeditor.cpp

```
void IconEditor::setPenColor(const QColor &newColor)
{
    curColor = newColor;
}
void IconEditor::setIconImage(const QImage &newImage)
{
    if (newImage != image) {
        image = newImage.convertToFormat(QImage::Format_ARGB32);
        update();
        updateGeometry();
    }
}
```

- La función `setPenColor()` define un nuevo color para el pincel.
- La función `setIconImage()` define la imagen a editar.
  - La llamada a `update()` fuerza un repintado del widget usando la nueva imagen.
  - La llamada `QWidget::updateGeometry()` informa al layout que contenga este widget que él ha cambiado su tamaño y qe adapte su tamaño al que tenga ahora el widget.

Un editor de iconos: chap05/iconeditor/iconeditor.cpp

```
void IconEditor::setZoomFactor(int newZoom)
{
    if (newZoom < 1)
        newZoom = 1;
    if (newZoom != zoom) {
        zoom = newZoom;
        update();
        updateGeometry();
    }
}
```

- La función `setZoomFactor()` establece un nuevo factor de zoom para la imagen.
- Las funciones `penColor()`, `iconImage()`, y `zoomFactor()` están implementadas como funciones inline en el fichero de cabecera.

Un editor de iconos: chap05/iconeditor/iconeditor.cpp

```
void IconEditor::paintEvent(QPaintEvent *event)
{
    QPainter painter(this);
    if (zoom >= 3) {
        painter.setPen(palette().windowText().color());
        for (int i = 0; i <= image.width(); ++i)
            painter.drawLine(zoom * i, 0,
                             zoom * i, zoom * image.height());
        for (int j = 0; j <= image.height(); ++j)
            painter.drawLine(0, zoom * j,
                             zoom * image.width(), zoom * j);
    }
    for (int i = 0; i < image.width(); ++i) {
        for (int j = 0; j < image.height(); ++j) {
            QRect rect = pixelRect(i, j);
            if (!event->region().intersect(rect).isEmpty()) {
                QColor color = QColor::fromRgba(image.pixel(i, j));
                painter.fillRect(rect, color);
            }
        }
    }
}
```

- La función `paintEvent()` se llama cada vez que el widget necesita repintarse:
  - Al aparecer por primera vez.
  - Cuando cambia su tamaño.
  - Al ocultar otra ventana que lo tapaba parcialmente.
- Por defecto, no hace nada, dejando blanco el widget.
- Podemos forzar un repintado con:
  - `Widget::update()`: Encola el repintado hasta que Qt procese los eventos de repintado.
  - `QWidget::repaint()`: Fuerza un repintado inmediato.
- Cada widget tiene asociada una paleta (`QPalette`) que se puede obtener con `palette()` (grupos de colores dependiendo del estado del widget: `Active`, `Inactive` o `Disabled`).

- En una paleta, podemos obtener la brocha (`QBrush`) usada para el texto (foreground general) con `windowText()`

Un editor de iconos: chap05/iconeditor/iconeditor.cpp

```
QRect IconEditor::pixelRect(int i, int j) const
{
    if (zoom >= 3) {
        return QRect(zoom * i + 1, zoom * j + 1, zoom - 1, zoom - 1);
    } else {
        return QRect(zoom * i, zoom * j, zoom, zoom);
    }
}
```

- La función `IconEditor::pixelRect()` se usa desde `paintEvent()` para construir un pixel aumentado con el zoom (un `QRect`).

Un editor de iconos: chap05/iconeditor/iconeditor.cpp

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

- Cuando el usuario pulsa un botón del ratón, el sistema genera un evento de ratón, y llama a la función virtual `mousePressEvent()`.

Un editor de iconos: chap05/iconeditor/iconeditor.cpp

```
void IconEditor::mouseMoveEvent(QMouseEvent *event)
{
    if (event->buttons() & Qt::LeftButton) {
        setImagePixel(event->pos(), true);
    } else if (event->buttons() & Qt::RightButton) {
        setImagePixel(event->pos(), false);
    }
}
```

- Los movimiento de ratón (con un botón del ratón pulsado) hacen que el sistema llame a la función virtual `mouseMoveEvent()`.

- Si quisiéramos que también se llame a la función anterior sin pulsar el botón del ratón usar: `QWidget::setMouseTracking()`

Un editor de iconos: chap05/iconeditor/iconeditor.cpp

```
void IconEditor::setImagePixel(const QPoint &pos, bool opaque)
{
    int i = pos.x() / zoom;
    int j = pos.y() / zoom;
    if (image.rect().contains(i, j)) {
        if (opaque) {
            image.setPixel(i, j, penColor().rgba());
        } else {
            image.setPixel(i, j, qRgba(0, 0, 0, 0));
        }
        update(pixelRect(i, j));
    }
}
```

- La función `setImagePixel()` se llama desde `mousePressEvent()` y `mouseMoveEvent()` para pintar o borrar un pixel.
  El parámetro `pos` es la posición del ratón en el widget.
- En el constructor de `IconEditor` hemos usado la llamada:

  `setAttribute(Qt::WA_StaticContents);`
  - Este atributo indica a Qt que el contenido del widget no cambia cuando cambia su tamaño al ser redimensionado, y que el contenido queda *anclado* a la esquina superior izquierda del widget.
  - Normalmente cuando se redimensiona un widget, Qt genera un evento de repintado para toda la parte visible del widget.
  - Con este atributo conseguimos reducir el repintado a la parte que aparezca nueva.