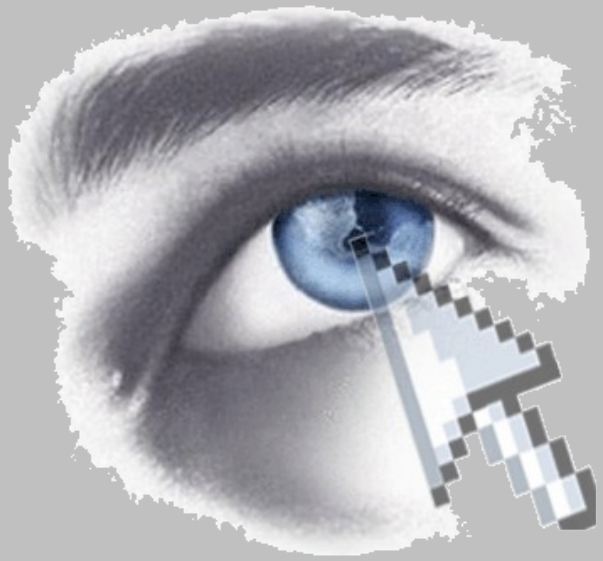


INFO : Interfaces Homme-Machine et Ergonomie du logiciel

Cours #4 : Programmation avec Qt5

Qt : Généralités

- Qt est une bibliothèque multiplateforme contenant :
  - Fonctions d'interfaces graphiques
  - Composants d'accès aux données
  - Connexions réseaux
  - Gestions de processus
- Devient ce qu'on appelle un FRAMEWORK :

  kit de composants logiciels structurels, qui sert à créer les fondations ainsi que les grandes lignes de tout ou d'une partie d'un logiciel
- Développée en C++ par la société Trolltech (Norvège), repris en 2008 par Nokia, puis cédé en 2012 à digia, utilisé pour l'OS du BlackBerry
- Sous licence mixte propriétaire / LGPL
- Disponible pour Windows, Unix et Mac OS

Qt : Généralités

- Déployable pour les platformes mobiles (Embedded Linux, Windows Mobile... Blackberry 10)
- Version Qt5, générateur d'interface rapide `Qt Quick`

Qt Simulator

Qt, les smarphones, les tablettes,etc

- Depuis la version 5.1, Qt propose le support de Qt sur Android, iOS et BlackBerry

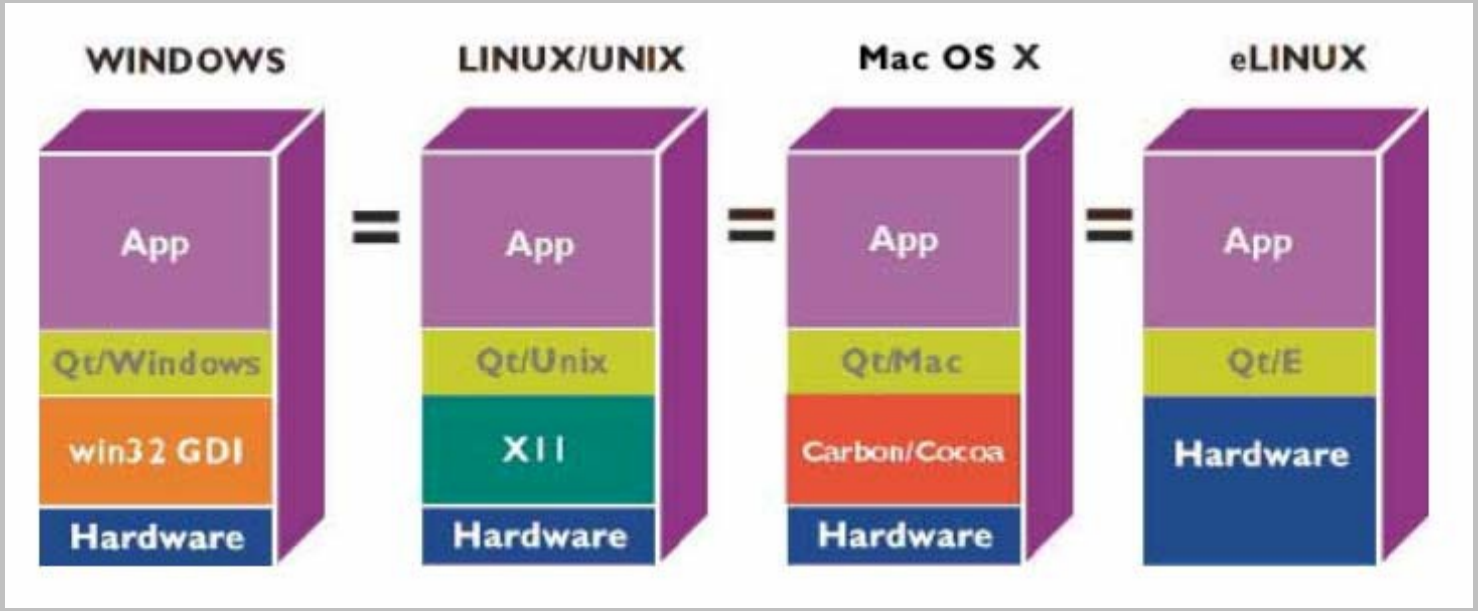
- Qt for Android enables you to run Qt 5 applications on devices with Android. A besoin de:
  - The Android SDK Tools (permet de créer des applications Java pour Android et un simulateur pour tester ses applications)
  - The Android NDK (permet de créer des applications natives C++ pour Android - application Java qui appelle un « module » écrit en C++)
  - OpenJDK v6 or later (l'environnement d'exécution de Java (JRE))
  - Apache Ant v1.8 or later
- Qt allows you to write advanced applications and UIs once, and deploy them across desktop and embedded operating systems without rewriting the source code saving time and development cost.

- En anglais, se prononce: kioute … (cute!)
- Qt 5.5 est sorti en octobre 2015:
  - « Qt is a C++ toolkit for multiplatform GUI application development. In addition to the C++ class library, Qt includes tools to make writing applications fast and straightforward. Qt's multiplatform capabilities and internationalization support ensure that Qt applications reach the widest possible market.
  - Qt's classes are fully featured to reduce developer workload, and provide consistent interfaces to speed learning. Qt is, and always has been, fully objectoriented.
  - The Qt C++ toolkit has been at the heart of commercial applications since 1995 (Adobe, IBM, Motorola, NASA, and Volvo, + smaller companies and organizations). »

- Qt est utilisé dans de nombreuses applications:
  - KDE
  - Google Earth
  - Skype
  - VLC
  - European Space Agency
  - DreamWorks
  - Lucasfilm,
  - Panasonic
  - Philips
  - Samsung
  - Siemens
  - En France, la freebox V6 a été développée avec QT, et un framework sera disponible pour créer ses applications (SDK QML Freebox sorti en 2015)

Qt : Couches logicielles

- Qt est une surcouche juste au dessus du graphique natif de la machine cible.



Philosophie objet de Qt

- Qt est basé autour du modèle d'objet de Qt. Cette architecture est ce qui rend Qt puissant et facile à employer.
  - classe **QObject**
  - outil moc (Meta-Object Compiler)

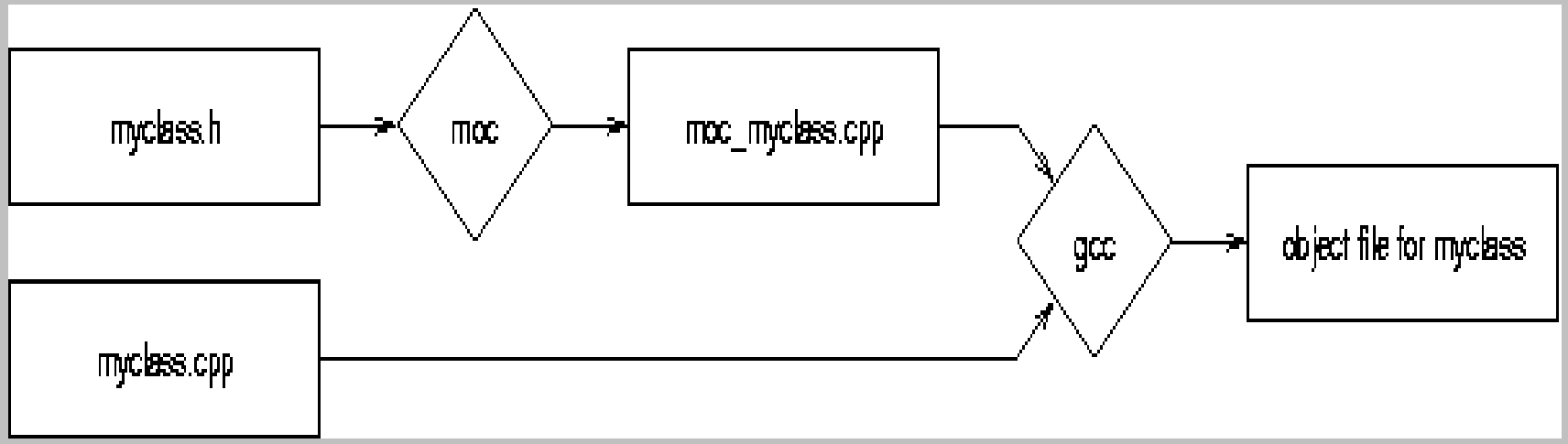
- En dérivant des classes de QObject un certain nombre d'avantages sont hérités :
  - Gestion facile de la mémoire.
  - Signaux et Slots.
  - Propriétés.
  - Introspection.

Fonctionnement de Qt

- Une application de Qt est 100% C++
- Signal et Slot sont des mots-clés : ils sont simplement remplacés par du C++ approprié par le préprocesseur (MOC)
- Les slots sont alors implémentés comme n'importe quelle méthode membre de classe tandis que les signaux sont implémentés par MOC
- Chaque objet tient alors une liste de ses connexions (quels slots sont activés par quel signal) et de ses slots (qui sont employés pour construire la table de connexions dans la méthode connect). Les déclarations de ces tables sont cachées dans la macro Q_OBJECT

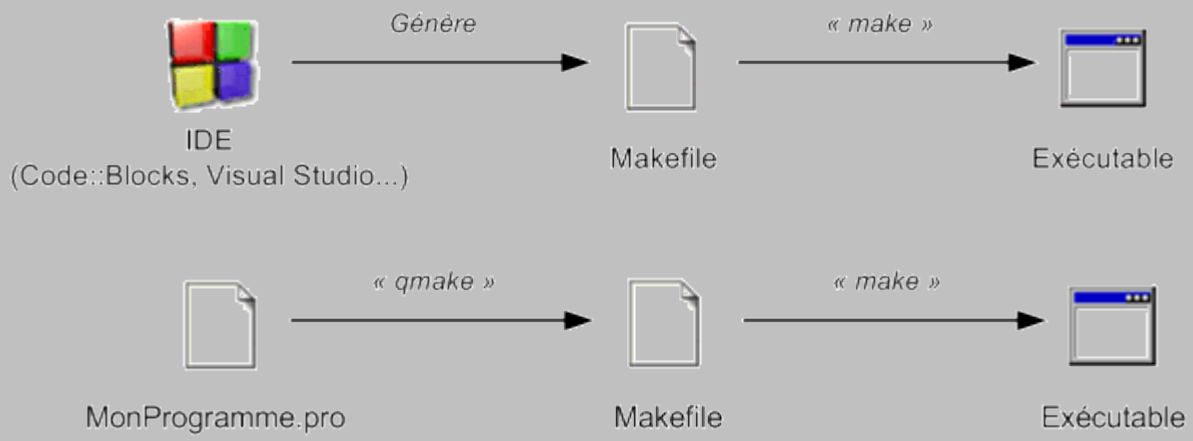
Dans la pratique

- **Idéalement**, chaque classe doit être déclarée dans un fichier d'en-tête **spécifique et séparé**.
- De même, l'implémentation d'une classe dérivée dans un fichier .cpp séparé est **conseillée**.
- Respect de la philosophie de l'implémentation de QT → code plus lisible et facilement maintenable.



Compilation

- La procédure de compilation utilise l'outil « qmake » qui permet de générer des MakeFiles à partir d'un fichier de projet « .pro »



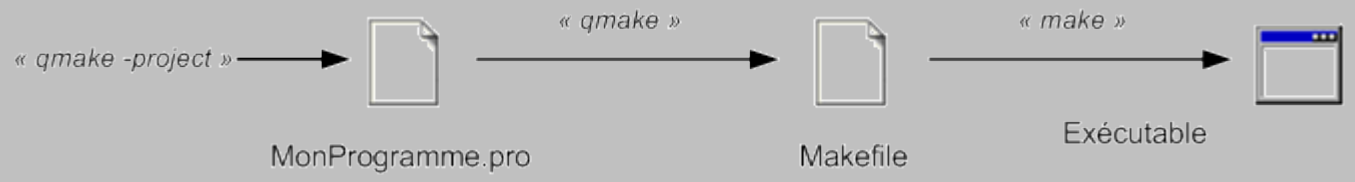
- Cet outil est particulièrement appréciable lorsque l'on développe un logiciel appelé à fonctionner sur plusieurs plates-formes (Linux et Windows par exemple), car cela évite d'avoir à maintenir des fichiers Makefile adaptés à chaque environnement.

Compilation

- Le fichier .pro peut être édité (« à la main »), dérivé de fichier .pro existant, ou pour des projets plus ambitieux, généré par des IDE (Integrated Development Environment) comme QTDesigner, QtCreator, …
  Ex: (fichiers main.cpp, hello.cpp et hello.h):

```
QT += widgets      #charger le module QtWidgets
HEADERS += hello.h
SOURCES += hello.cpp
SOURCES += main.cpp
```

- Qt peut générer un .pro automatiquement avec qmake avec l'option –project. qmake va analyser les fichiers du dossier et générer un fichier .pro basique



qmake –project -> .pro qmake –o Makefile -> Makefile make -> .exe

Gestion de la mémoire

- Lorsque l'on crée une instance d'une classe héritée de QObject il est possible de passer un pointeur vers l'objet parent.
- Ainsi, lorsque le parent est supprimé, les objets fils le sont également facilitant ainsi grandement la gestion de la mémoire.

```
class monObjet : public
QObject
{
Public :
  MonObjet(QObject
*parent=0, char *name=0) :
QObject(parent, name)  {
…
}
} ;
```

Exemple

- Classe dérivée de QObject qui affiche tout ce qui lui arrive sur la console.

```
class VerboseObject : public QObject
{
public:
  VerboseObject(QObject *parent=0, char *name=0) :
QObject( parent, name )
  {
    std::cout << "Created: " << QObject::name() << std::endl;
  }

  ~VerboseObject()
  {
    std::cout << "Deleted: " << name() << std::endl;
  }

  void doStuff()
  {
    std::cout << "Do stuff: " << name() << std::endl;
  }
};
```

Application Qt minimale

```
#include <Qapplication>
#include <QPushButton>

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

  QPushButton hello("Hello world!");



  hello.resize(100, 30);



  hello.show();



  return app.exec();
}
```

Headers de Qt

Création de l'application

Création d'un bouton

Redimensionnement

Affichage du bouton

Lancement de l'application et attente de la fin

Signaux et slots

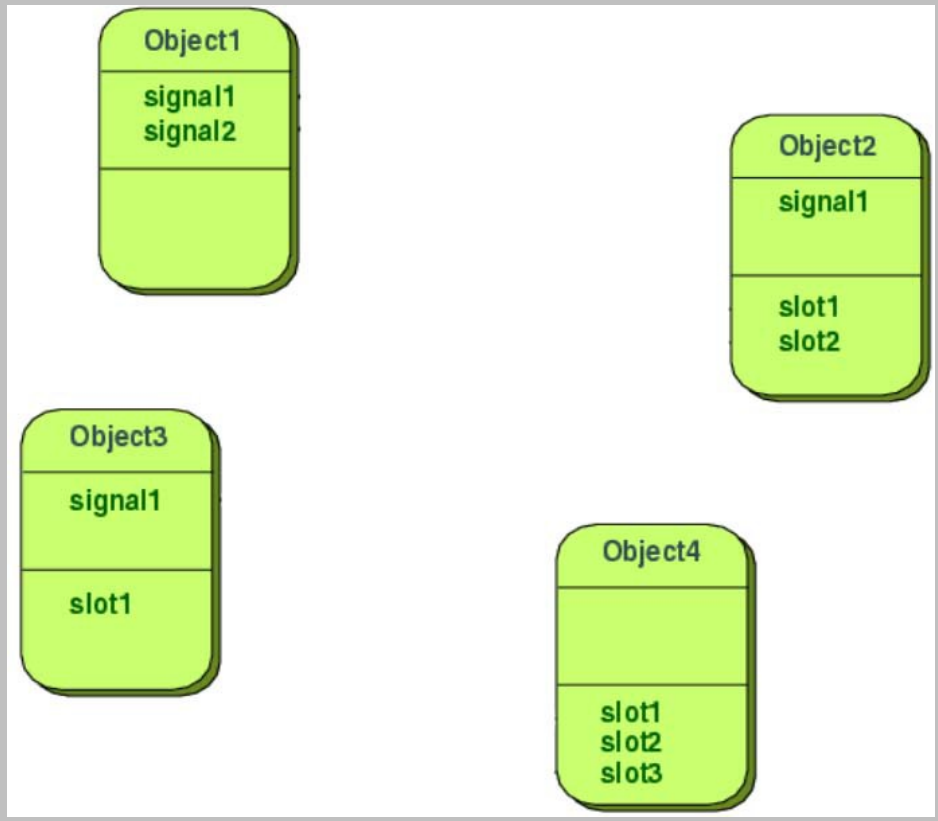
- Qt propose une manière extrêmement élégante et souple pour associer **des fonctions reflexes** aux **événements** : Le concept de signaux (ou événements) et de slots (ou fonctions ou *Event Handlers*)
- Les signaux et slots fournissent un mécanisme permettant de librement associer des traitements à un ou plusieurs événements. En associant un slot à un événement (ou signal) on permet d'appeler automatiquement le slot à chaque occurrence de l'événement ou des événements correspondants.
- L'émetteur du signal ne sait rien du récepteur et réciproquement. **Ceci rend le couplage émetteur-récepteur faible** et donc facilité la réutilisation et l'indépendance de chacun.

Signaux et slots

- Un signal est émis lorsqu'un événement particulier se produit : perte de focus, clic, etc…
- Qt fournit un ensemble d'événements prédéfinis, mais il est possible par héritage ou par ajout de code de définir ses propres événements.
- En réponse à un signal (lui-même généralement produit à suite à un événement), une fonction (méthode de l'objet) de type slot est appelée. Là également, un certain nombre de slots sont prédéfinis (quit() par exemple), mais il est possible de proposer ses propres slots.
- **Couplage faible :**
  - **Passage de paramètres standardisé**
  - **Communication « asynchrone », pas d'accusé de réception**

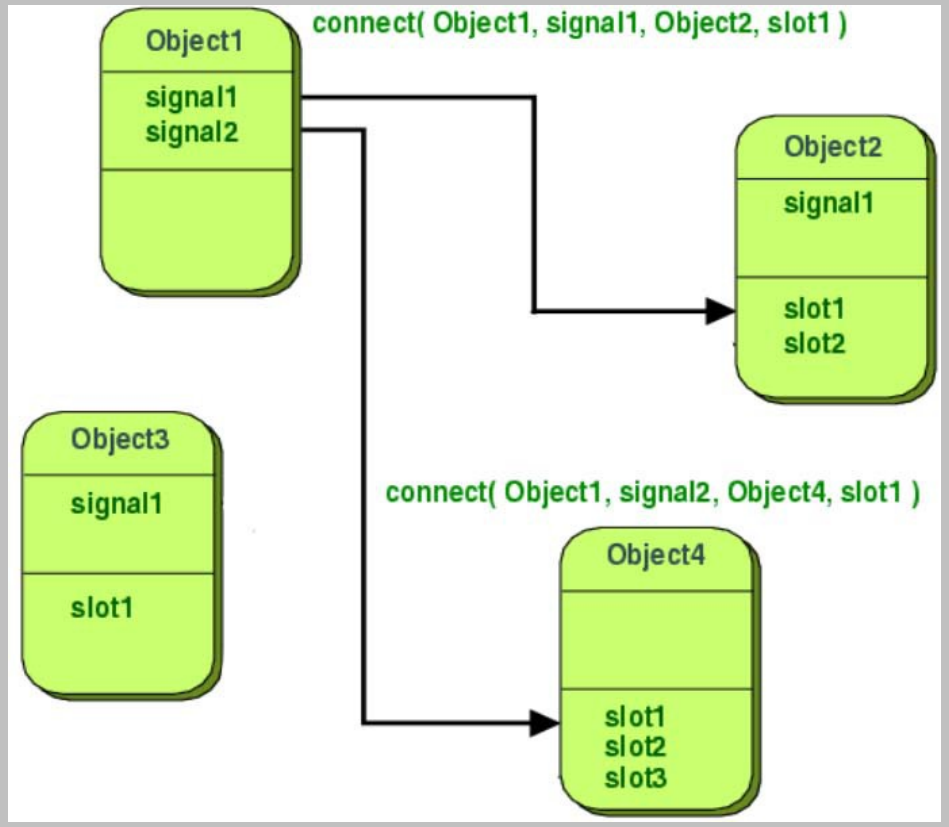
Signaux et slots

- Les objets (essentiellement les widgets) peuvent posséder des signaux, des slots ou les deux



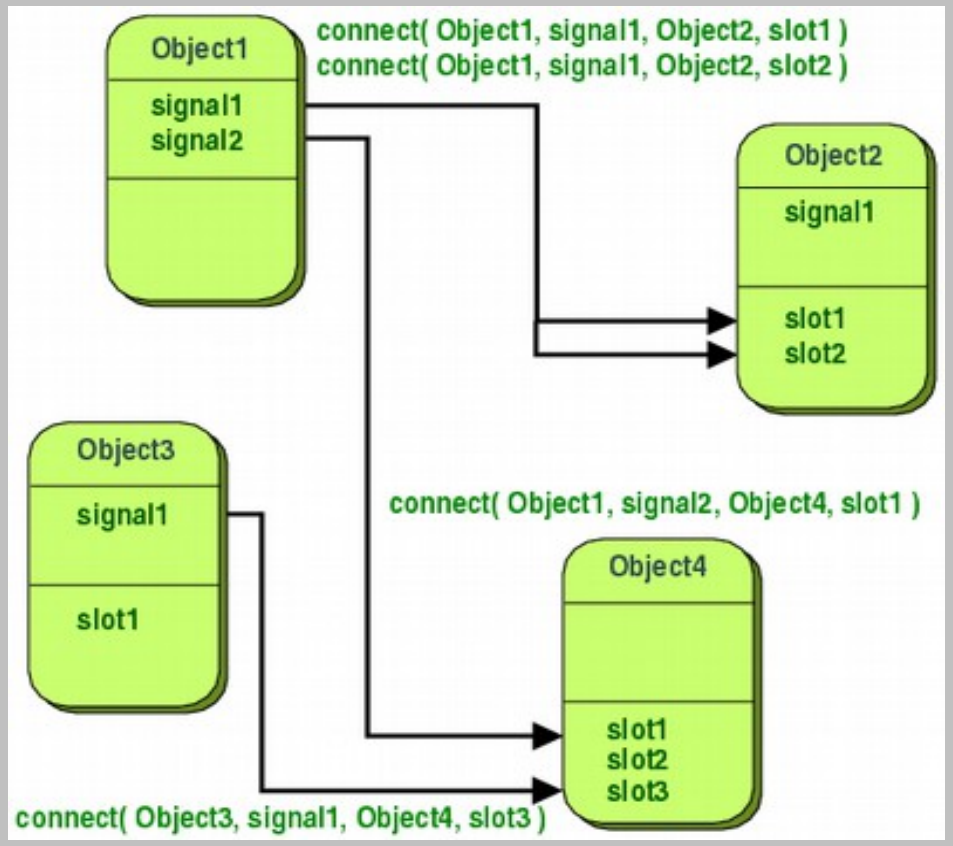
Signaux et slots

- On utilise une directive connect pour associer un slot à un signal



Signaux et slots

- Il est bien sûr possible de connecter deux slots à un même signal
- Exécution séquentielle



Emission de signaux

- On peut émettre des signaux à tout moment
- Souvent, c'est en réaction à l'action d'un utilisateur = surcharge des méthodes (virtuelles) hérités de QWidget :

```
virtual void keyPressEvent ( QKeyEvent * e )
virtual void keyReleaseEvent ( QKeyEvent * e )
virtual void leaveEvent ( QEvent * event )
virtual void mouseDoubleClickEvent ( QMouseEvent *
e )
virtual void mouseMoveEvent ( QMouseEvent * e )
virtual void mousePressEvent ( QMouseEvent * e )
virtual void mouseReleaseEvent ( QMouseEvent * e )
... etc
```

Attention au focus !

Emission de signaux depuis le code

```
class myWidget : public QWidget
{
  Q_OBJECT
  ...
  signals: void mysignal (int);
  ...
};

void myWidget::keyPressEvent(QKeyEvent *event)
{
  int count;
  switch (event->key())
  {
  case Qt::Key_Left:
    ... // do something
    break;
  case Qt::Key_Space:
    count ++;
    emit(mysignal(count))
    break;
  default:
    ...
}
```

Signaux et slots : Pratique

- Définir un signal / slot, émettre un signal, relier signal et slot
- Dans .h

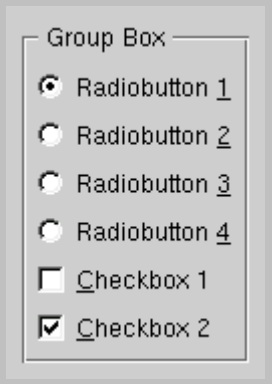
```
class myClass : public Qobject
{
  Q_OBJECT
  ...
  signals: void event ();
  ...
  public slots: void action ();
  ...
};
```

- Dans .cpp

```
...
emit(evenementSurvenu());
...
connect(myClass,SIGNAL(event()),myClass,SLOT(action
()));
```

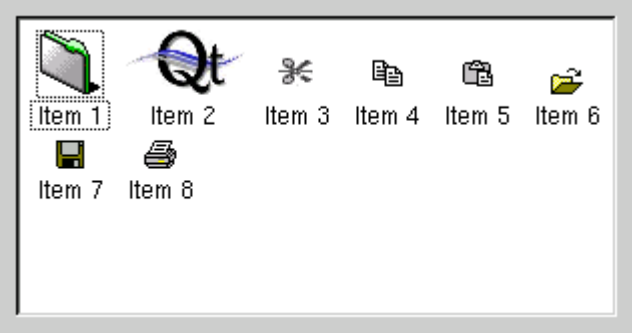
Quelques widgets de base

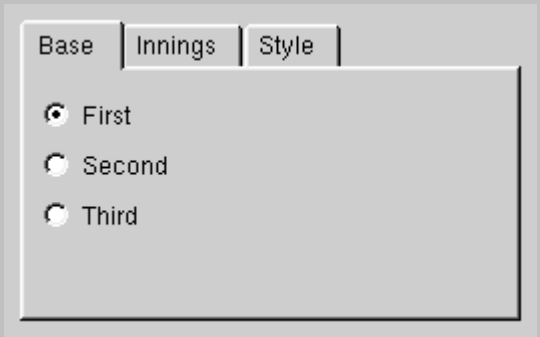
- Buttons et dérivés :
  - Pushbuttons(QPushButton): boutons classiques pouvant contenir un texte ou une image (pixmap). Il peut également avoir un comportement de bouton poussoir(togglebutton).
  - Checkboxes(QCheckBox): cases à cocher
  - RadioButtons(QRadioButton): idem précédent, mais un seul bouton peut être coché.
  - Lorsque l'on utilise des boutons radio, il est fréquent de ne vouloir qu'un seul bouton coché à la fois. Pour faire cela, il est nécessaire de regrouper les boutons concernés dans un QButtonGroup
- Evénements pour buttons et dérivés :
  - pressed: bouton pressé.
  - released:bouton relâché
  - clicked:bouton pressé puis relâché

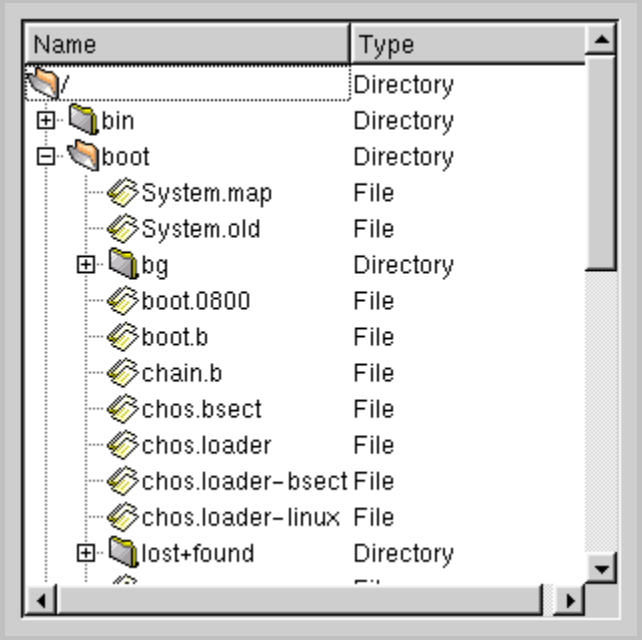


Quelques widgets de base

- Widgets pour tous usages…



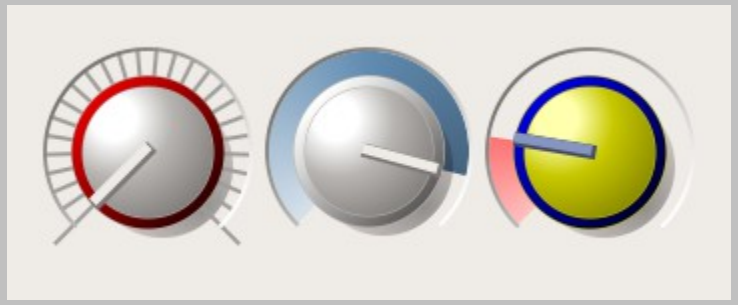




L'intérêt de la POO pour les GUI

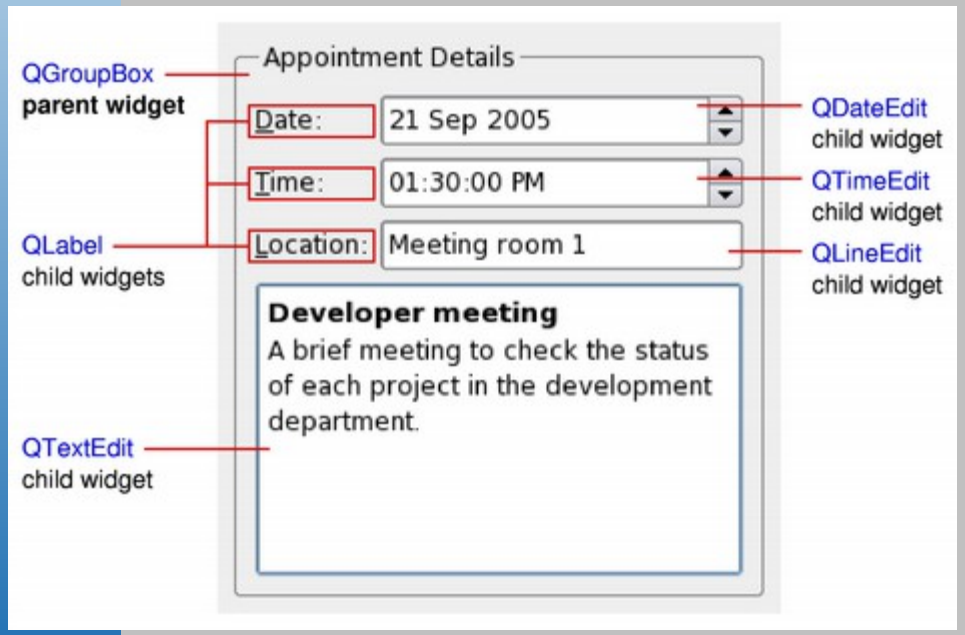
- L'apparence graphique de QDial est le suivant :

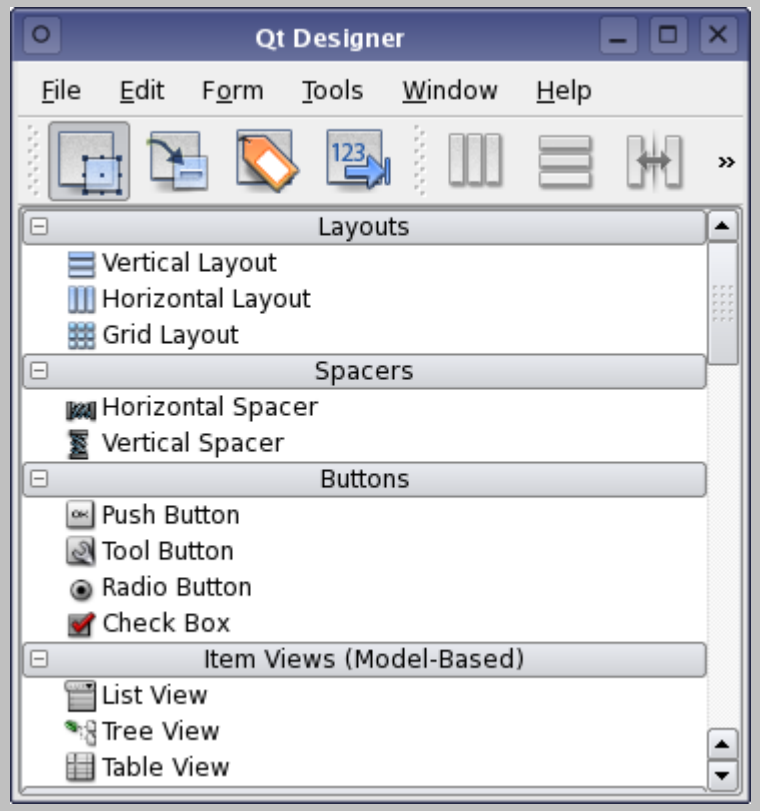
- On veut changer l'apparence, mais aussi le comportement
  - Changement graduel jusqu'à la valeur cliqué
  - Ajout d'un seuil
  - …
- Re-écriture complète !



Création de widgets

- Création dans le code, ou comme « ressources »: DEMO
- Utilisation de QtDesigner





Les widgets container ou geometry management

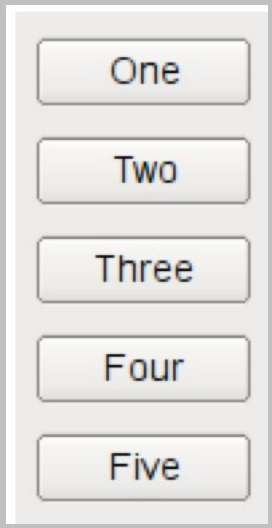
- Arrangement spatial de widgets
- A spécifier comme widget parent

QBoxLayout	Lines up child widgets horizontally or vertically
QButtonGroup	Organizes QButton widgets in a group
QGLayoutIterator	Abstract base class of internal layout iterators
QGrid	Simple geometry management of its children
QGridLayout	Lays out widgets in a grid
QGroupBox	Group box frame with a title
QHBox	Horizontal geometry management for its child widgets
QHBoxLayout	Lines up widgets horizontally
QHButtonGroup	Organizes QButton widgets in a group with one horizontal row
QHGroupBox	Organizes widgets in a group with one horizontal row
QLayout	The base class of geometry managers
QLayoutItem	Abstract item that a QLayout manipulates
QLayoutIterator	Iterators over QLayoutItem
QSizePolicy	Layout attribute describing horizontal and vertical resizing policy
QSpacerItem	Blank space in a layout
QVBox	Vertical geometry management of its child widgets
QVBoxLayout	Lines up widgets vertically
QVButtonGroup	Organizes QButton widgets in a vertical column
QVGroupBox	Organizes a group of widgets in a vertical column
QWidgetItem	Layout item that represents a widget

Les widgets container ou geometry management

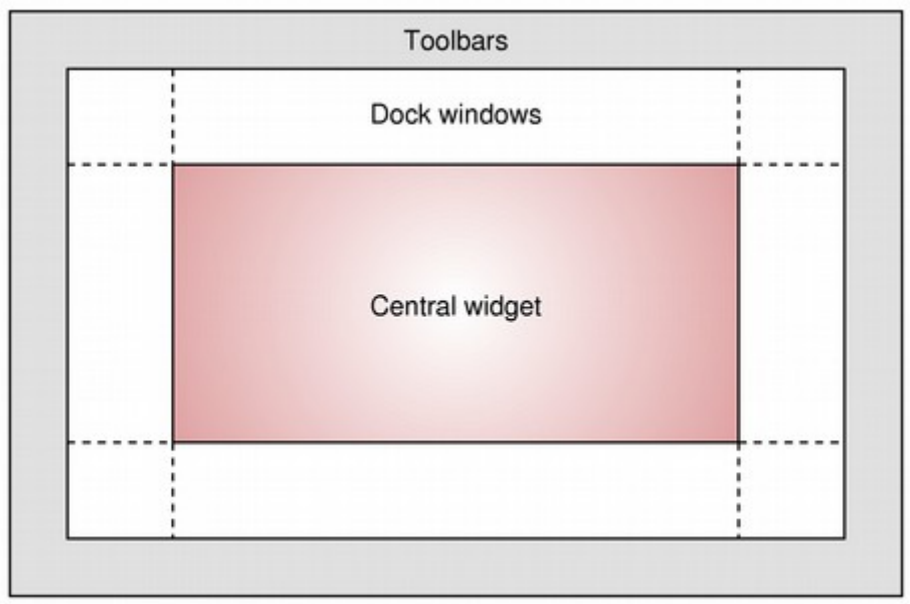
- Arrangement spatial de widgets
- A spécifier comme widget parent

```
QWidget *fenetre = new QWidget;
QPushButton *bouton1 = new QPushButton("One");
QPushButton *bouton2 = new QPushButton("Two");
QPushButton *bouton3 = new QPushButton("Three");
QPushButton *bouton4 = new QPushButton("Four");
QPushButton *bouton5 = new QPushButton("Five");

QVBoxLayout *layout = new QVBoxLayout;
layout->addWidget(bouton1);
layout->addWidget(bouton2);
layout->addWidget(bouton3);
layout->addWidget(bouton4);
layout->addWidget(bouton5);

fenetre->setLayout(layout);
fenetre->show();
```



Fenêtre principale pour application

- **QMainWindow** est la classe standard pour construire une application
- **QDockWidget** correspond à un widget pour créer des palettes d'outils (détachables) ou de zone d'aide.
  They can be moved, closed, and floated as external windows.
- **QToolBar** correspond à un widget générique pour y mettre des boutons d'action :
  - Barre d'outils, menus, etc...



Classe QMainWindow

- En général, on dérive une classe correspondant à l'application de QMainWindow. La mise en place des menus, barres d'outils, … est effectué dans le constructeur.
- Remarquer l'usage de QAction (*abstract user interface action that can be inserted into widgets*)
- QAction permet l'ajout de raccourcis clavier, de tooltips (messages d'aide), et l'utilisation de connexions signaux/slots pour déclencher une action.

```
ApplicationWindow:: ApplicationWindow(QWidget *parent) :
QMainWindow(parent)
{
...
newAct = new QAction(tr("&New"), this);
newAct->setShortcut(tr("Ctrl+N"));
newAct->setStatusTip(tr("Create a new file"));
connect(newAct, SIGNAL(triggered()), this, SLOT(newFile()));
...
}
```

Classe QMainWindow

- Après avoir créé les actions, ces objets abstraits sont insérés dans les widgets disponibles au sein de la classe QMainWindow, par exemple, dans une barre de menu.
- Remarquez l'usage de la methode menuBar() :

*QMenuBar * QMainWindow::menuBar () const*

Returns the menu bar for the main window. This function creates and returns an empty menu bar if the menu bar does not exist.

```
fileMenu = menuBar()->addMenu(tr("&File"));
fileMenu->addAction(newAct);
fileMenu->addAction(openAct);
...
fileMenu->addSeparator(); ...
```

Barre de menu

- QAction permet d'associer des actions indifféremment à des boutons d'une toolbar ou à des articles de menus.
- Le passage par QAction n'est pas nécessaire, mais conseillée.

```
// création de la partie déroulante du menu
QMenu *fileMenu = menuBar()->addMenu(tr("&File"));

// insertion des articles
fileMenu->addAction(tr("&Save"),this, SLOT(save()));
```

QString QObject::tr (const char * sourceText, const char * comment) [static]

Returns a translated version of sourceText, or sourceText itself if there is no appropriate translated version

-> UNICODE

-> Localization.

Menu : ouverture d'un fichier

- Il faut bien sur implémenter les SLOTs correspondants. Examinons de plus près l'ouverture d'un fichier.
- La connexion suivante a été faite :

```
connect(openAct,SIGNAL(triggered()),this,SLOT(open()));
```

Dans la définition de classe :

```
public slot:
    void ApplicationWindow::open()
```

Implementation de la méthode :

```
void ApplicationWindow::open()
{
fileName = QFileDialog::getOpenFileName(this,
tr("Open a File"),"..//..");
if (!fileName.isEmpty())
// do something
}
```

Utilisation de dialogues

- On fait l'usage de boites de dialogue « prédéfinies »
- Toute bonne bibliothèque fourni un ensemble de dialogues standards, le plus simple étant le dialogue d'information

QColorDialog	Dialog widget for specifying colors
QDialog	The base class of dialog windows
QErrorMessage	Error message display dialog
QFileDialog	Dialog that allow users to select files or directories
QFontDialog	Dialog widget for selecting a font
QInputDialog	Simple convenience dialog to get a single value from the user
QMessageBox	Modal dialog with a short message, an icon, and some buttons
QPrintDialog	Dialog for specifying the printer's configuration
QProgressDialog	Feedback on the progress of a slow operation

```
QString text=QInputDialog::getText("input","Enter your name",
QString::null, &ok, this );
```

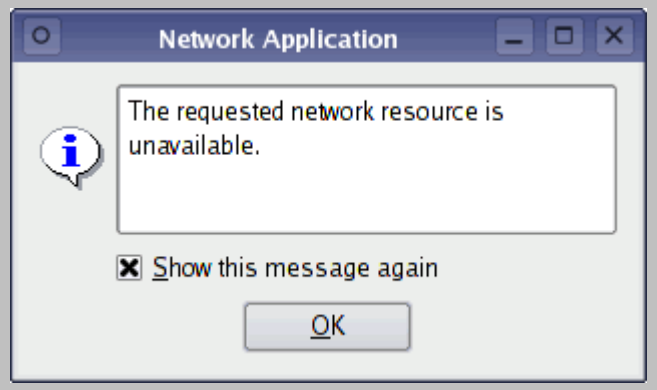
Exemple: QErrorMessage

- Toutes dérivent de QDialog
- Constructeur/Destructeur :

```
QErrorMessage ( QWidget *
   parent = 0 )
~QErrorMessage ()
```

- Affichage (c'est un SLOT, mais la méthode peut être appelée directement, pas seulement via connect)

```
void showMessage ( const
   QString & message )
```



Exemple avec retour d'information

- QMessageBox are small modal pop-up widgets used to display information , warning or critical information to user in a simple way

```
switch( QMessageBox::warning( this, .
"Application name",
"Could not connect to the server.\n" ,
"Try again", "Quit", 0, 0, 1 )
{
    case 0: // Try again or Enter
        break;
    case 1: // Quit or Escape
        break;
}
```

StandardButton QMessageBox::warning (QWidget * parent, const QString & title, const QString & text, StandardButtonsStandardButtons buttons = Ok, StandardButton defaultButton = NoButton) [static]

Opens a warning message box with the specified title and text. The standard buttons are added to the message box. defaultButton specifies the button used when Enter is pressed. defaultButton must refer to a button that was given in buttons. If defaultButton is QMessageBox::NoButton, QMessageBox chooses a suitable default automatically.
Returns the identity of the standard button that was clicked. If Esc was pressed instead, the escape button is returned.
If parent is 0, the message box is an application modal dialog box. If parent is a widget, the message box is window modal relative to parent

Dialogues standards

- QColorDialog

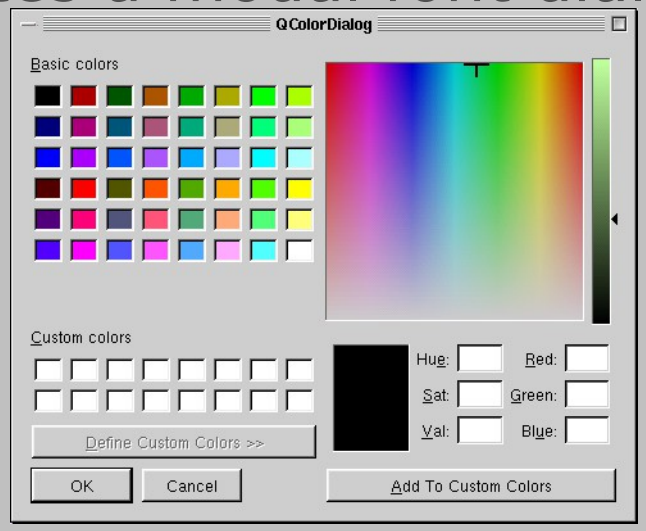
```
QColor QColorDialog::getColor ( const QColor & initial = Qt::white,
    QWidget * parent = 0 )
```

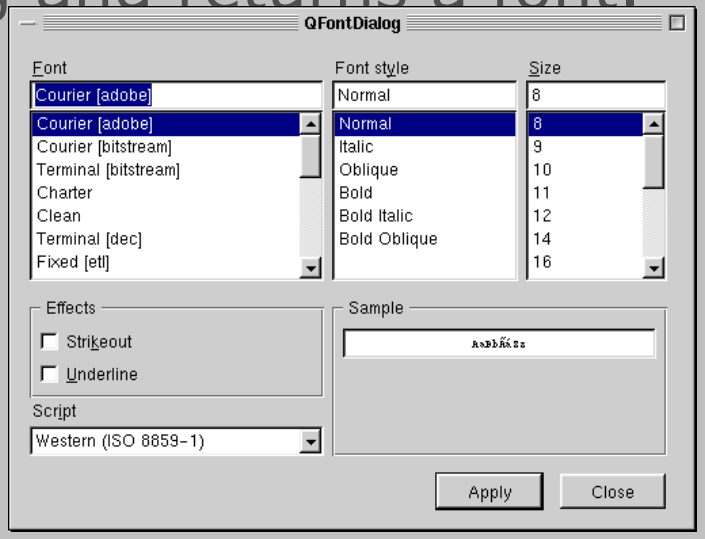
Pops up a modal color dialog, lets the user choose a color, and returns that color.

- QFontDialog

```
QFont QFontDialog::getFont ( bool * ok, const QFont & initial,
    QWidget * parent = 0 )
```

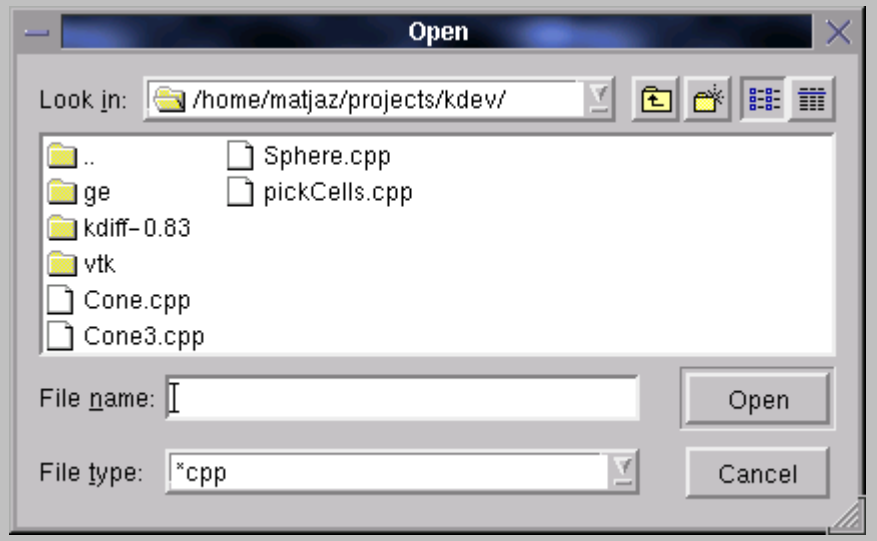
Executes a modal font dialog and returns a font.





QFileDialog

- Boite de dialogue standard pour parcourir une arborescence et choisir un fichier.
- The easiest way to create a QFileDialog is to use the static functions. On Windows, these static functions will call the native Windows file dialog, and on Mac OS X these static function will call the native Mac OS X file dialog.




```
QString s =
 QFileDialog::getOpenFileName(
    this,
    "Choose a file",
    "/home",
    "Images (*.png *.xpm
*.jpg)"
    );
```

Entrées/Sorties avec Qt

- Evidemment, respectant la notion de FRAMEWORK, Qt propose aussi des classes pour les entrées/sorties, par exemple la classe Qfile

```
QFile file ("in.txt");
if (!file.open(QIODevice::ReadOnly |
QIODevice::Text))
   return;
while (!file.atEnd())
{
   QByteArray line = file.readLine();
   process_line(line);  // par exemple
avec sscanf(...)
}
```

- Ainsi que de classes pour le parsing HTML / XML, les commandes SQL, l'usage des sockets TCP/UDP...

Lecture d'un fichier

- En enchainant les opérations, cela donne :

```
void ApplicationWindow::open()
{
  Qstring fileName = QFileDialog::getOpenFileName(
    this,
    tr("Open a File"),
    "..//..");
  if (!fileName.isEmpty())
  {
    QFile file(fileName);
    if (!file.open(QIODevice::ReadOnly | QIODevice::Text))
      return;
    while (!file.atEnd())
    {
      QByteArray line = file.readLine();
      process_line(line); // par exemple avec sscanf(...)
    }
  }
}
```