Formation Qt

Gilles Maire

2018

Plan de la formation

Présentation

Rubriques

- Histoire de Qt
- Rappels C++

Histoire de Qt

L'histoire de cute Qt

- **1993-1994** Haavard Nord et Eirik Chambe-Eng fondent Trolltech en Suède
- **Qt1 :** 1996 Sortie de la librairie (Qt = cute = joli)
- **KDE :** 1996 Matthias Ettrich lance Kde s'appuyant sur Qt
- **Qt2 :** 1999 avec version systèmes embarqués
- **Qt3 :** 2001 Mac OSX + expressions régulières
- **Qt4 :** 2005 Modules
- **Qt5 :** 2013 QtQuick et QML support Android

Qt de sociétés en sociétés

- **1994 :** Société Quasar technologie rebaptisée TrollTech
- **2009 :** Trolltech est rachetée par Nokia (qt.nokia.com)
- **2012 :** Nokia cède Qt à Digia qui annonce le portage sous Android et IOS (www.qt.io)
- **2012 :** naissance de la fondation qt-project.org pour la version de Qt Open Source (qt-project.org)

Par le passé : une jungle de produits

- **Qt Jambi :** portage Qt en Java (abandonné par nokia en 2009) supporté communautairement en 4.7.0
- **Necessitas :** projet Qt pour Android indépendant de Google et Nokia (intégré dans la 5.0)
- **Maemo :** Plate-forme de développement pour tablette et portable à base de Qt4 (depuis 2009)
- **Meego :** fusion de Maemo et de Moblin (Linux Intel) (février 2010)
- **Tizen.org :** (2011) OS Intel Samsung HTML5 2014 un smartphone avec Tizen (modèle Z) sans cesse retardé
- **Extended :** plate-forme Nokia pour Embbeded fusionnée en Qt4.4 avec Qt4-SDK
- **Qt Quick :** inclut QML/ Javascript $=>$ Qt5 $+$ Web
- **QTopia :** ancien nom du projet QtExtended
- **PyQt :** Qt pour Python. Portage de la librairie en environnement Python

Aujourd'hui : un seul environnement

- **Qt Creator :** un environnement
- **QtLinguist :** aide à l'internationalisation
- Les librairies fournies sous forme de binaires ou de sources
- La librairie source unique pour le mode Embedded, le mode Linux, MacOsX, Windows.
- Versions open source sur le site http://www.qt-project.org
- Disponible avec un compilation C++ GNU et MSC++ sous Windows
- Sous Linux et MacOsX via le compilateur Gnu C++

Rappels C++

C++ Rappel : définition de classe

- Fichier classe.h

```
#ifndef CLASSE_H
#define CLASSE_H
#include <ClasseMere>
class Classe : public ClasseMere {
public:
    Classe();
    ~Classe();
private:
    int m_element ; // variable
    void unemethode( int valeur) ;
// valeur est facultatif mais pas int
};
#endif // CLASSE_H
```

C++ : CPP associé

- Fichier classe.cpp

```
#include "classe.h"
Classe::Classe(){

}
Classe::~Classe()
{

}
void Classe::unemethode( int entier) {
}
```

C++ : parfois les deux en un

- Dans classe.cpp

```
Classe::Classe();
// tout d'un coup
class MasousClasse: public Classe
{
public :
   void NouvelleMethode( int valeur){
   methode ( valeur ) ;
   }
} ;
// souvent utilisé pour dériver une classe protected
```

Rappels C++ Encapsulation

- **Private :** accessible qu'à l'intérieur de la classe ou aux fonctions externes déclarées friend
- **Public :** accessible à l'intérieur et à l'extérieur de la classe
- **Protected :** comme private mais accessible par les classes filles uniquement
- **Explicit :** pas d'initialisation de constructeur de transtypage
- **Liste d'initialisation :** uniquement sur les constructeurs

```
MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),ui(new MainWindow)
```

- **fonction statique :** n'est pas visible dans la classe mais utilisable via class::fonction()

Initialisation de variables

```
int Nom1 ; Nom1 = valeur ;
int Nom2 = valeur ;
int Nom3 (valeur) ; // équivalent
int Nom4(valeur),Nom5(valeur);
int &Nom6(valeur) // référence Nom6 vaut valeur
int Nom7(); // HORREUR
int const Nom8 (valeur) ; // ne peut changer de valeur
```

Exemples réels :

```
int i=3;
int j(2);
```

QtCreator

Rubriques

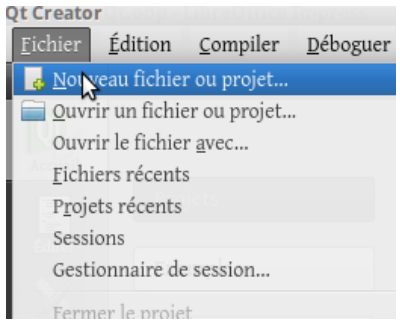
- Prise en main
- L'aide
- Aide Freenode
- Usages

Prise en main

QtCreator

- est un éditeur C++ Qt avec complétion
- Il fournit une aide
- Il permet de compiler et de débuguer
- Il permet de créer un projet avec des modèles
- il fournit une interface graphique de conception des interfaces graphiques
- Il gère SVN et Git
- Il permet plusieurs modes de compilation (librairies statiques, librairies dynamiques)

Ouverture de projet / Modèles



- Nouveau projet Application Graphique (desktop ou Embedded)
- Nouveau projet Application graphique pour portable (Symbian ou Maemo)
- Nouveau projet console (le mode console pour faire un programme C++ sans interface graphique mais utilisant le SDK)

Projets Quick et autres modèles

- Qt Quick (QML)
- Autres projets :
  - librairie Qt,
  - fabrication de Widget,
  - test avec QtTest,
  - plugin de Qtcreator
- De base projets Qt via qmake
- Projets non QtCreator (C++)
- Projets cmake (ouvrant un fichier cmakelist)

Vérification

- Découverte :
- Lancer QtCreator
- Créer un nouveau projet Qt Application graphique
- Compiler / Exécution
- En mode Projets explorer les différents fichiers
- Cliquer sur le formulaire
- Modifier la fenêtre en ajoutant n'importe quel objet sur la gauche

(par exemple un bouton)

- Recompiler

Le mode console (Linux)

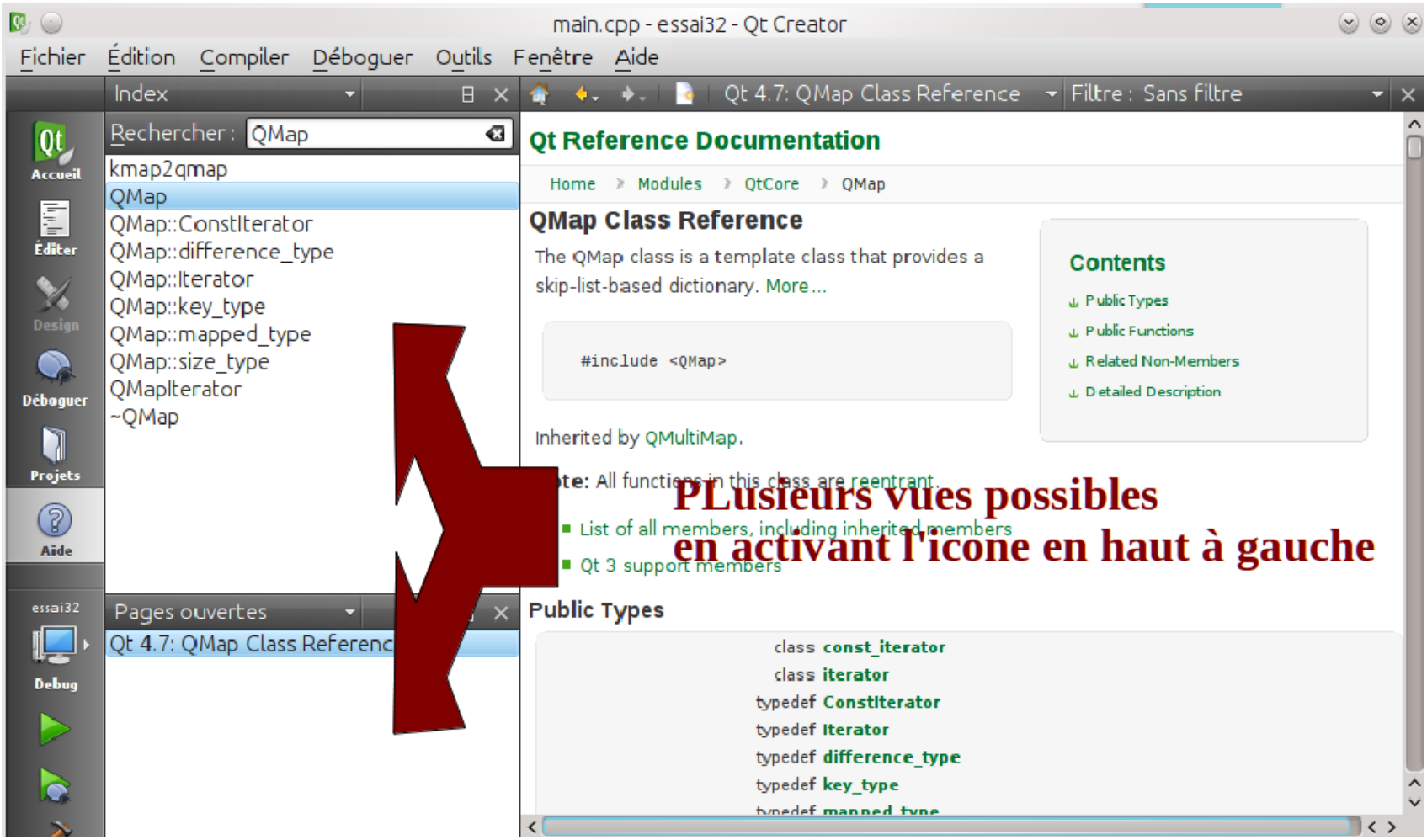
- En mode console sous l'exercice précédent on peut lancer les programmes suivants :
  - make clean
  - make distclean
  - qmake
  - make
- Sous Windows on ne dispose pas de ces commandes de bas niveau

Barre des tâches

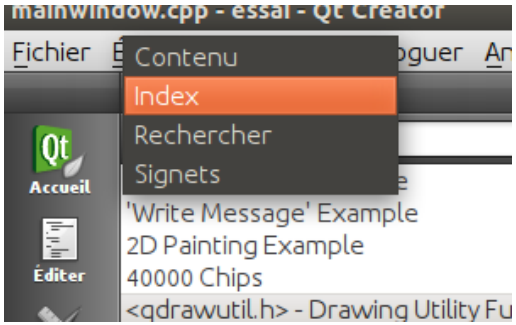
- Accueil
- Éditeur
- Designer
- Débugueur
- Définition paramètres projet
- Aide
- Choix mode compilation
- Exécution (et compilation)
- Exécution en mode debug
- Compilation

L'aide

L'aide : Qt assistant intégré



L'onglet d'aide



- Contenu : présentation de l'aide organisée par chapitre
- Index : recherche par classe, méthodes, signaux etc
- Rechercher : recherche par un mot clé
- Mécanisme de signets
  - ajouter un signet fenêtre centrale à gauche zone numéros de ligne
  - Menu Outil/Signet Suivant Précédent
  - Mode signet dans le combo (Projet,Documents ouverts etc..)
- On peut ajouter des vues d'aide ou en retirer

Aide autre

- Depuis l'éditeur
  - complétion
  - F1 sur les mots clés
  - Passer d'un objet à sa déclaration et vice versa F2
- Sur les concepts plus compliqués regarder les exemples dans la page d'accueil
- Aller dans la partie designer pour voir les propriétés dans la partie droite

Quelques conseils pour lire une page d'aide

- Une page d'aide concerne une classe. Lire la partie introductive qui présente des exemples
- La classe dont on consulte l'aide est héritée d'une classe mère qui possède des méthodes auxquelles on a accès. En conséquence quand on découvre une classe :
  - découvrir les méthodes des classes mères
  - découvrir les signaux des classes mères
- Souvent le nombre de méthodes est impressionnant mais on sait que l'on cherche un retour de type entier ou QString. Il faut donc balayer la liste des méthodes en s'occupant que de la colonne correspondant au type recherché.

Aide Freenode

L'aide sur IRC/Freenode

- Nous allons apprendre à nous servir de Freenode qui contient deux canaux importants d'aide concernant Qt :
  - `#qt` : en langue anglaise
  - `#qt-fr` : en langue française
- Si votre logiciel ne mémorise pas les salons on peut se rendre dans un salon par

```
/join #qt-fr
```

- Nous allons apprendre à nous servir du logiciel XChat qui fonctionne sous Linux et Windows mais il existe un grand nombre de logiciels
- Nous allons également voir les règles et les usages à respecter quant au réseau freenode

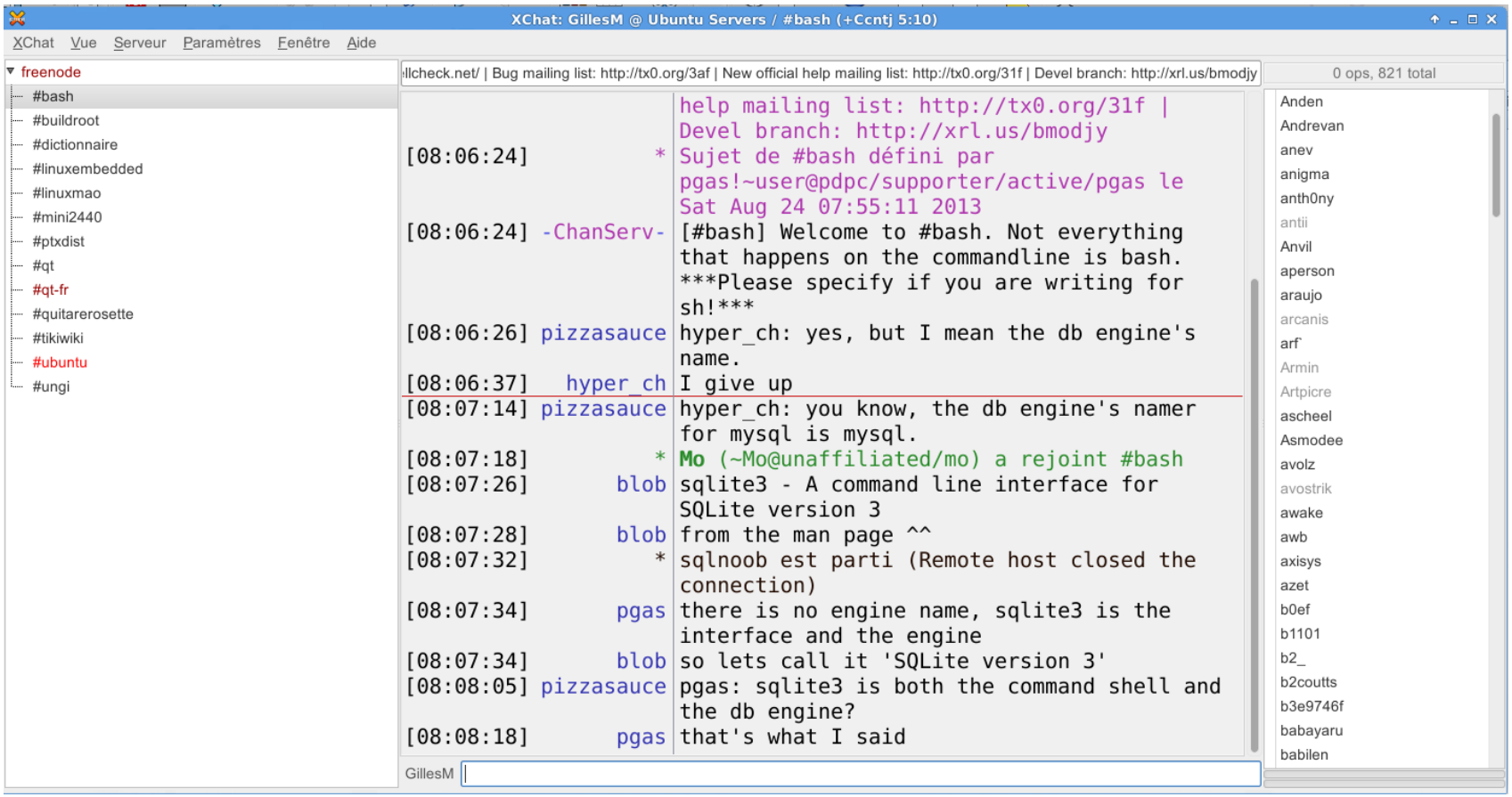
L'aide par IRC

- IRC est un protocole de chat qui est utilisé pour l'entraide dans les développements des logiciels libres
- Les réseaux IRC techniques Open Source se trouvent principalement sur Freenode pour la partie généraliste, mais on trouve également des serveurs IRC consacrés au langage Perl, à la distribution Debian (OTFC), aux Bitcoins
- L'utilisation d'IRC vient à remplacer les forums qui ont tendance à ne plus s'adresser qu'à des utilisateurs très novices.
- Nous allons apprendre à nous servir d'IRC et à respecter les usages en pratique sur ce type de réseau

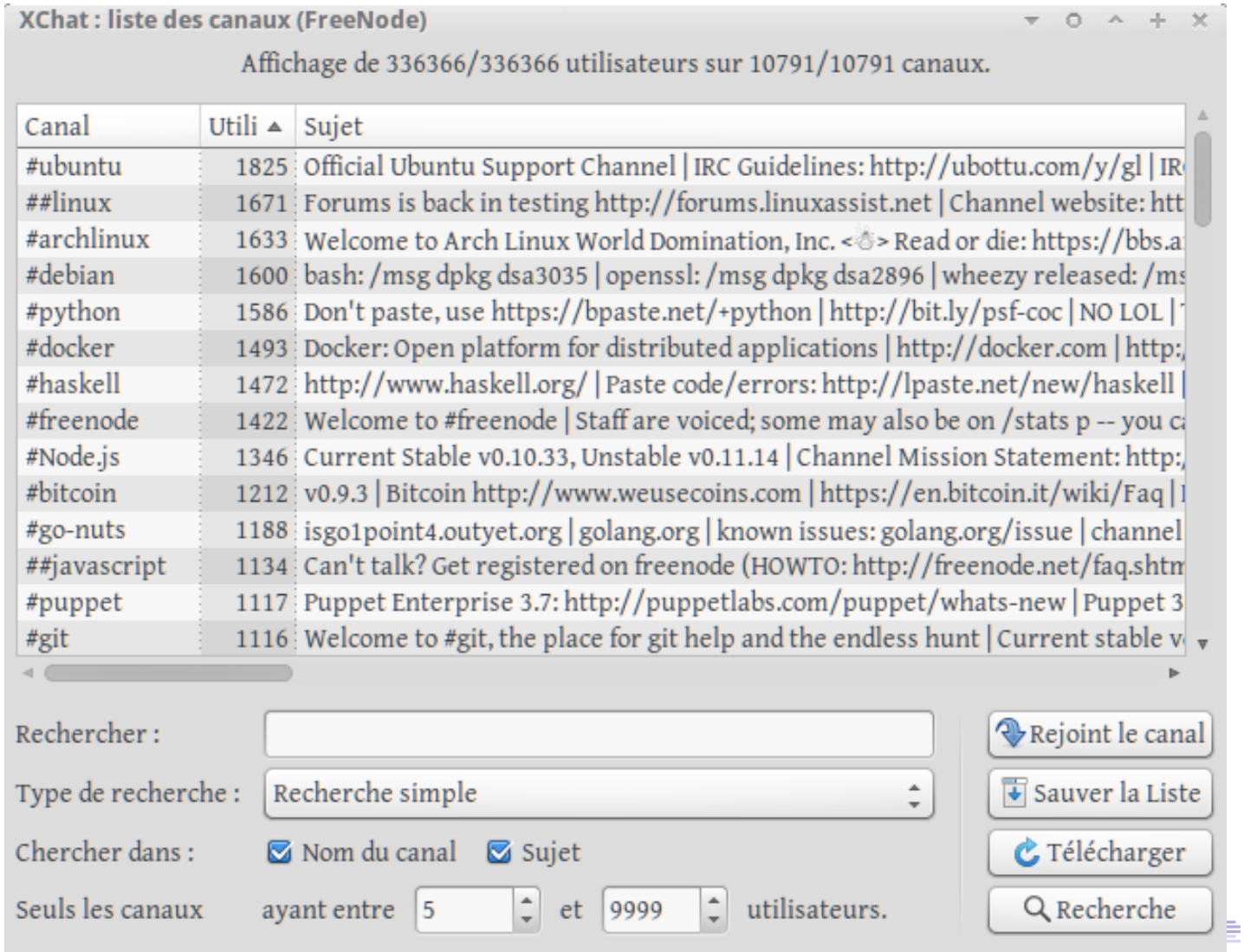
Logiciels IRC

- Logiciels spécialisés
  - On se connecte à Freenode par un logiciel spécialisé IRC (port 8001) : Linux(XChat, Quassel, Hexchat, Konversation, KVirc, Pidgin) Windows (mirc)
  - Ces logiciels, gardent la trace des précédentes discussions permettant de retrouver des informations contenues dans d'anciennes discussions, ils affichent la liste des canaux de discussions.
- On trouve pour certains réseaux notamment freenode une interface web http://webchat.freenode.net
  - qui nécessite de connaître le canal sur le quel on veut se connecter car la passserelle n'en fournit pas la liste.
  - c'est une interface qui passe par les ports 80 ce qui peut s'avérer pratique si on ne peut utiliser qu'un port http à cause d'un Firewall.

Exemple de canal



Liste des canaux



IRC : Présentation du réseau Freenode

- **Freenode :** réseau de type IRC
  - 300 000 utilisateurs
  - plus de 10 000 canaux répartis par logiciel
- **Serveur :** irc.freenode.net
- **Port :** 8001
- Protection des mots de passe par mécanisme NICKSERV
- Utilisé comme canal d'aide par beaucoup de logiciels Open Sources.
- Passerelle Web à http://webchat.freenode.net

Usages

Freenode : Nickserv

- Nickserv un service permettant de réserver son pseudo sur Freenode

```
/msg NickServ REGISTER password youremail@example.com
```

- Ensuite pour se connecter :

```
/msg NickServ IDENTIFY monnick password
```

- La connexion automatique se fait dans l'option de connexion du logiciel IRC

Quelques règles de bon sens

- Les personnes connectées ne sont pas obligées de vous répondre mais en général si elles sont devant leur écran elles vous répondront si elles connaissent la réponse à votre question
- À part si le canal se termine par fr ou si le topic (le sujet du salon) est en français la langue principale est l'anglais
- Ne vous présentez pas, n'indiquez pas que vous avez une question, posez la.
- Il est interdit d'envoyer des codes sources qui pollueraient l'affichage. Pour cela on utilise pastebin
- Le sujet de certains salons comprend des instructions précédées d'un point d'exclamation. Il s'agit de commandes robot qui vous donnent des premières aides

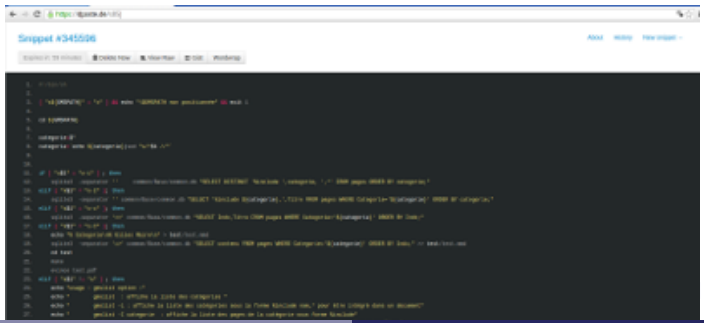
```
!getting_help
<knoba> GillesMM: "getting_help" : before asking your question, read the
!relevant_logs and !showconfig factoids, and prepare a single pastebin
containing all of that data. if you don't understand what this means,
or if you need help doing this, please let us know.
also see !pastebin
```

Pastebin

- Pastebin est un service Web permettant de copier du code pour ne pas polluer les fenêtres des salons
- Il est directement accessible depuis certains environnements de développement.
- Dans tous les cas, vous copiez les sources qui apparaîtront formatées dans le service pastebin ; une adresse URL vous est communiquée que vous indiquez dans IRC.
- Si vous accédez à pastebin par le web l'adresse est au choix : dpaste.org, fpaste.org ou pastebin.ca
- Le résultat à communiquer vous sera renvoyé par exemple

`http://pastebin.ca/3PYeZEgL`

- Les sites pastebin vous propose de conserver la trace pendant une durée de temps paramétrable. Pour des raisons de sécurité, ne laissez pas traîner à vie des fichiers de configuration qui pourraient finir sur google.



Imagebin

- Pour communiquer une copie d'écran explicitant un problème ou une solution vous utilisez imagebin à l'adresse : http://imagebin.org
- Sur le même principe que pastebin vous récupérez l'adresse de votre image que vous communiquez sur IRC
- Il est efficace de mettre des pastilles indiquant les zones que vous voulez souligner.(voir l'application gimp http://www.gillesmaire.com/tiki-index.php?page=draw-numbers)
- Dans l'exemple suivant, on montre aux développeurs de l'application Ardour un problème avec quelques pastilles numérotées via imagebin.org

Les canaux freenode spécialisés C++/qt

- En langue anglaise :
  - **#qt :** aide générale sur un canal de 500 personnes parmi lesquelles le développeurs digia
  - **#qt-creator :** plutôt réservé au développement et au bug qu'à l'utilisation
  - **#qt-labs :** les développeurs du projet Qt-project comptant environ 200 personnes
  - **#qt-webkit :** concerne les question sur la partie WebKit de Qt
  - **##c++ :** canal dédié à l'aide sur C++ (700 utilisateurs)
  - **##c++-basic** : canal des question basiques sur C++
- En langue française :
  - **qt-fr** : 30 à 50 personnes

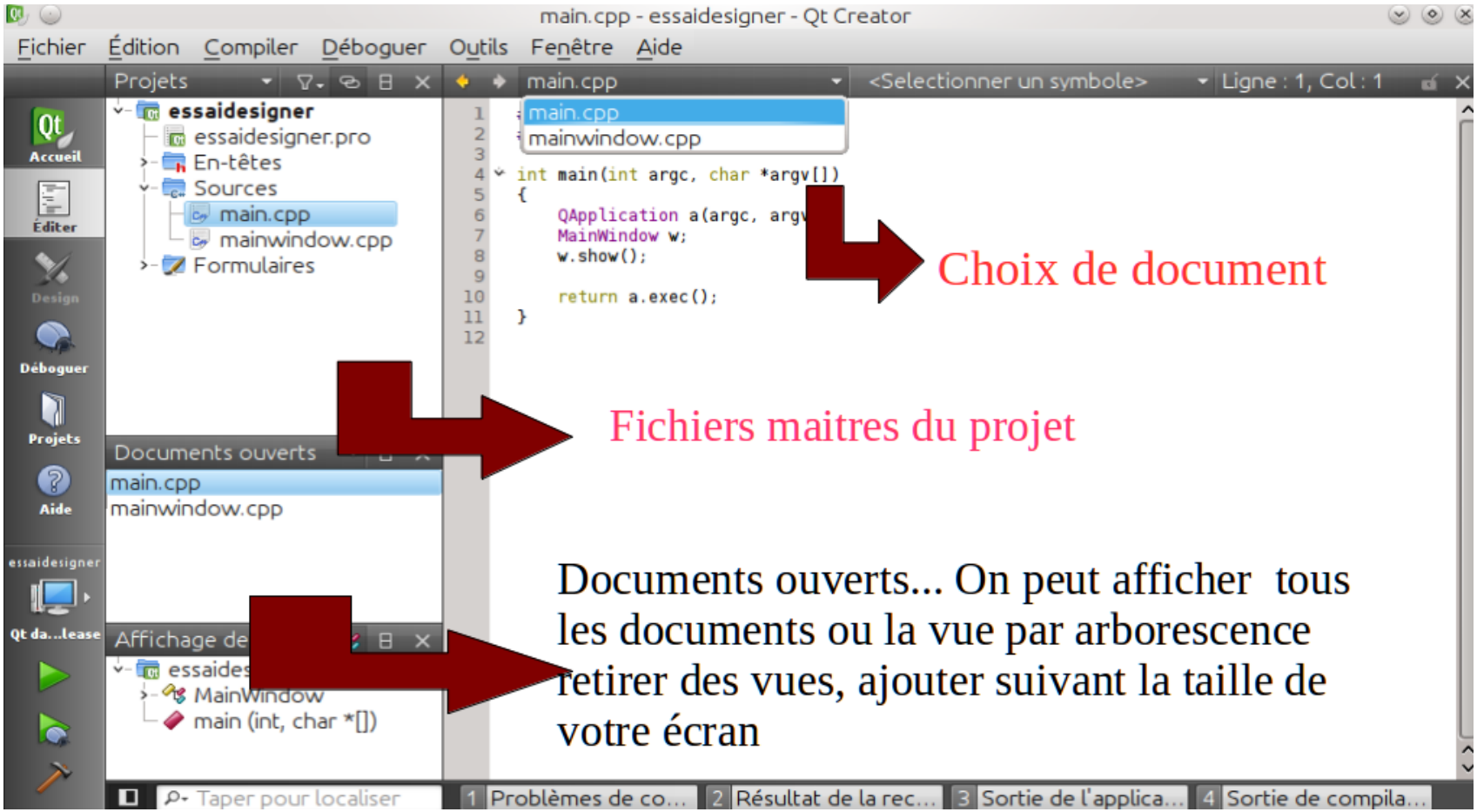
Utilisation

Rubriques

- Édition de texte
- Les fichiers en présence
- fichier.pro

Édition de texte

Éditeur de texte



Caractéristiques de l'éditeur

- Complétion
- Fonction *Édition/Recherche*
  - Sur le fichier courant ou les fichiers du projet
  - Sur plusieurs projets
- Ouverture fermeture d'un bloc de code
- Détermination -> ou . sur objet en fonction de pointeur ou d'instantiation
- Click droit item *Refactoriser*
  - permet de déclarer une fonction définie ou de de définir le squelette d'une fonction déclarée
  - permet à partir d'un attribut déclarer setters et getters
- Scinder les fenêtres dans le menu *fenêtre*
- Les flèches de navigation en haut à gauche de l'éditeur permettent de retrouver les endroits u code où l'on était précédemment positionné
- SHIFT CONTROL / sur des lignes surlignées passent en commentaires
- trois / devant un nom de méthode dans le .h insère un commentaire Doxygen

Quelques raccourcis clavier

- Accessible et modifiable par le menu Outils/Options/Environnement/Clavier
- CTRL B : Buid - CTRL R : Run
- CTRL W : fermer une fenêtre - CTRL MAJ W : fermer toutes les fenêtres
- CTRL ++ CTRL- - : agrandir réduire la taille de la police de caractères
- CTRL J : joindre deux lignes
- MAJ F4 : basculer entre design (édition d'UI) et source

Les fichiers en présence

Les différents fichiers d'un projet

- **Fichier.pro :** permet de générer un fichier Makefile
- **main.cpp :** programme principal

```
#include "mainwindow.h
#include <QApplication>
int main(int argc, char *argv[])
{
    QApplication a(argc, argv); // voir plus loin
    MainWindow w; // voir dans mainwindow.h
    w.show();     // méthode show montre la fenêtre
    return a.exec(); // attend la fin de l'application
}
```

Fichier mainwindow.h

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H
#include <QMainWindow>

namespace Ui {class MainWindow;} // Ui remplace MainWindow
class MainWindow : public QMainWindow
{ // MainWindow dérive de QMainWindow
    Q_OBJECT // Macro définissant un MetaObjet
public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();
private:
    Ui::MainWindow *ui;// pointe les objets de MainWindow
};
#endif // MAINWINDOW_H
```

Fichier mainwindow.cpp

```
#include "mainwindow.h"
#include "ui_mainwindow.h"
// ui_maintwindow.h est la traduction en C++
// du design de l'interface
//constructeur
MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow) // ui est initialisé
{
    ui->setupUi(this);// setup de l'interface
}
//destructeur
MainWindow::~MainWindow()
{
    delete ui;
}
```

Fichier mainwindow.ui

- fichier xml définissant l'interface dessinée à la souris
  - à ne pas modifier à la main
  - ne présente pas d'intérêt didactique
- Permet la génération d'un fichier ui_mainwindow.h qui construit l'interface en C++
  - ce fichier est situé dans le répertoire où sont générés les objets
  - ce fichier est assez didactique car il permet de comprendre l'utilisation des objets Qt quand la doc n'est pas claire

fichier moc_mainwindow.cpp

- ce fichier est le fichier de définition des objets déclarés par la macro Q_OBJECT
- Quand on ajoute Q_OBJECT dans un fichier de déclaration il faut refaire un qmake pour générer ce fichier
- Ce fichier ne doit bien sûr pas être modifié.

Fichier Makefile

- sert à la construction du projet
- est généré par qmake à partir du fichier nomprojet.pro.
- make comprend les arguments
  - clean
  - distclean
  - all
  - diverses étiquettes dont install (qui n'est pas implémenté)
- Ce fichier ne doit pas lui non plus être modifié manuellement

fichier.pro

qmake

- Le programme qmake transforme le fichier pro en Makefile, son appel est automatisé par QtCreator
- Contenu fichier pro

```
QT       += core gui widgets
TARGET = essai
TEMPLATE = app
SOURCES += main.cpp mainwindow.cpp
HEADERS  += mainwindow.h
FORMS    += mainwindow.ui
```

qmake détail

```
TEMPLATE = app lib subdirs
DEFINES +=  ...  ( -D option)
DESTDIR = ../../bin
HEADERS +=
INCLUDEPATH = inclusion de header d'autres librairies
LIBS = autres librairies
RESSOURCES= nous verrons plus loin
CONFIGS += console qt x86
QT += core gui widgets network opengl phonon sql svg xml  webkit
TRANSLATIONS = fichiers de traductions
```

Gestion de la compilation

- L'onglet projet permet de définir l'environnement de compilation
- Suivant qu'on positionne le mode Debug ou Release on inclut ou on n'inclut pas les symboles de débugage
- C'est le qmake appelé qui définit l'environnement en fonction de la variable CONFIG positionnée à DEBUG ou RELEASE CONFIG+=debug
- *shadow* permet de créer un répertoire indépendant des sources et de laisser le répertoire propre

Objets Qt

Rubriques

- Classes de base
- Classes présentes dans le Designer
- QLineEdit
- Slots et signaux

Classes de base

Les principales classes

- **QObject :** objet de plus bas niveau
- **QCoreApplication :** c'est l'application en mode console (sans fenêtre)
- **QApplication :** c'est l'application en mode graphique
- **QWidget :** l'objet de base de tous les objets graphiques
- **QMainWindow :** c'est la fenêtre principale montrée au démarrage par le designer
- **Différents objets :** zones de saisie, étiquettes, listes déroulantes

QCoreApplication

- **QCoreApplication :** intègre la partie application vis à vis de l'OS
- *addLibraryPath*, *applicationDirPath*, *applicationFilePath*, *applicationName*, *applicationPid*, *applicationVersion*, arguments etc
- comprend *exec()* qui entre dans une boucle qui attend un exit()
- **InstallTranslator :** installe les fichiers de traduction
- signal **aboutToQuit() :** appelé quand on quitte l'application

QApplication

- **QApplication :** intègre la partie graphique
- Hérite de *QCoreApplication* en ajoutant des méthodes orientées window
- initialisation des palettes de couleurs
- Initialisation des feuilles de style
- initialisation des Polices de caractères
- initialisation du curseur de la souris

Premiers programmes

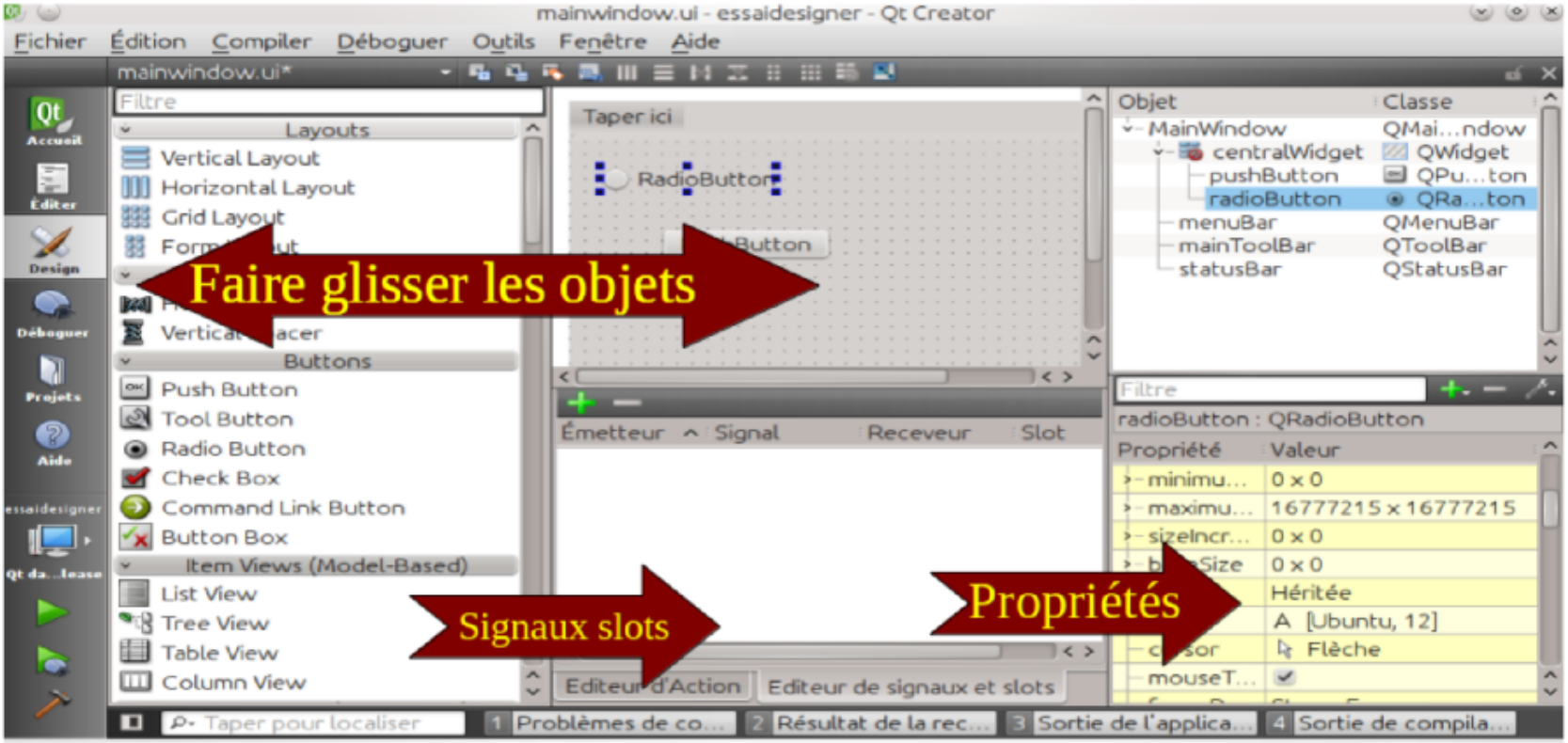
- Créer un programme minimum en QApplication
- Créer un programme minimum en QCoreApplication
- Comparer les fichiers générés

Classes présentes dans le Designer

Présentation de la partie designer

- Si une classe en C++ se matérialise par un fichier cpp et le fichier h de définition, une classe graphique se voit ajouter un fichier d'extension ui
- Ce fichier ui est un fichier XML que l'on peut manipuler au moyen de la partie designer de Qtcreator
- Le fichier *ui* est converti par qtcreator en un fichier h qui est en fait un fichier cpp et qui a pour nom `ui_fichier.h`
- Il ne faut surtout pas modifier ce fichier ni même tenter de toucher au fichier xml `ui`, sous peine de ne plus pouvoir le modifier en mode designer

Designer


mainwindow.ui - essaidesigner - Qt Creator
Fichier Édition Compiler Déboguer Outils Fenêtre Aide
mainwindow.ui*
Filtre
Layouts
Vertical Layout
Horizontal Layout
Grid Layout
Buttons
Push Button
Tool Button
Radio Button
Check Box
Command Link Button
Button Box
Item Views (Model-Based)
List View
Tree View
Table View
Column View
Taper ici
RadioButton
Faire glisser les objets
Signaux slots
Propriétés
Émetteur Signal Receveur Slot
Editeur d'Action
Editeur de signaux et slots
Objet Classe
MainWindow QMainWindow
centralWidget QWidget
pushButton QPushButton
radioButton QRadioButton
menuBar QMenuBar
mainToolBar QToolBar
statusBar QStatusBar
radioButton : QRadioButton
Propriété Valeur
minimum 0 x 0
maximum 16777215 x 16777215
sizeIncr 0 x 0
Héritée
Flèche
Taper pour localiser
Problèmes de co...
Résultat de la rec...
Sortie de l'applica...
Sortie de compila...

QMainWindow

- Elle est composée de menus QMenu
- La barre d'outils (QToolBar) : fenêtre à petites icones entre le menu et la partie centrale
- Elle comprend la partie centrale QWidget qui est le type générique de tous les objets graphiques
- QStatusbar : bande horizontale du bas avec les status
- Elle est appelée généralement dans le programme main.cpp par la méthode show() mais elle peut être masquée par hide()
- La partie attente de fin d'exécution de la fenêtre étant effectuée par le exec() du programme main()

QMenu

- composés d'Actions (QAction) comprenant
  - **QIcon :** une Icone
  - **QShortcut :** un raccourci claviers
  - **QToolTip :** note d'aide courte
  - **QWhathis :** note plus détaillée
  - **QData** pour transporter plus d'informations
- les actions (QAction) des menus sont munis de signaux :
  - **changed :** modification de l'action
  - **toggled :** l'action passe d'actif à non actif ou réciproquement
  - **hovered :** la souris passe sur l'action
  - **triggered :** l'action est demandée

Les objets d'une interface

- **QPushButton :** bouton Ok, annuler, bouton action
- **QRadioButton :** oui / non , homme / femme (QButtonGroup exclut plusieurs QRadioButton, s'obtient par multi sélection bouton droit)
- **QCheckBox :** marié/Célibataire
- **QComboBox :** liste déroulante
- **QLineEdit :** ligne de saisie
- **QListView, QTableView :** liste et table utilisable avec un modèle
- **QListWidget, QTableWidget :** liste et table de widgets (label, case à cocher, combobox)
- **QLabel** (sert souvent à afficher des images) mais surtout les étiquettes informatives en face des QLineEdit
- **QTextBrowser** et **QWebView** : classe éditeur de texte et afficheur de pages HTML

QLineEdit

Exercice : QLineEdit

- Construire à l'aide du designer un programme dont l'interface

comprend
un QLineEdit Nom et un QLineEdit Prenom

- avec une valeur par défaut dans chaque ligne
- Mettre une étiquette Nom et Prénom devant chaque champ (par un

QLabel)

Exercice : QLineEdit (suite)

- Ajouter un bouton, cliquer droit sur ce bouton demander “Aller au slot" puis “clicked” une fonction on_pushButton_clicked sera créée
- Dans la fonction ainsi créée
  - rendre grisé le *Qlinedit prenom*
  - mettre une valeur au champ prenom
- NB : on accède au linedit via la variable ui

Barre de sélection de mode et layout

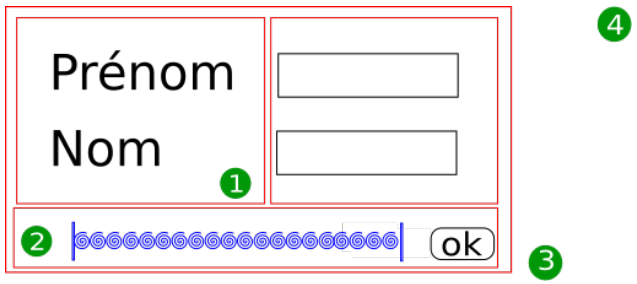
1. Permet de revenir au mode édition de Widget quand on est passé au mode 2,3 et 4
2. Passage au mode éditeur de signaux et de slots visuels
3. Passage en mode Buddy (voir après)
4. Passage en mode définition de l'ordre des passages d'un objet à l'autre par une tabulation (cliquer chaque objet dans l'ordre, double click recommence)
5. Positionnement d'un layout horizontal
6. Positionnelent d'un layout vertical
7. Positionnement d'un splitter horizontal
8. Positionnement d'un splitter vertical
9. Positionnement dans un layout de formulaire
10. Positionnement dans une grille
11. Suppression d'un layout
12. Sélection de la taille optimale

Mode buddy

- En utilisant dans le nom d'un QLabel le caractère &, on définit ainsi un raccourci clavier qui sera activé en mode buddy
- On peut associer ce racourci à priori inutile à un autre objet (par exemple un QLineEdit)
- On passe en mode Buddy dans lequel toutes les étiquette réagissent en devenant rouge
- On peut alors en draguant la souris vers l'objet associé lier les deux objets.
- Le raccourci clavier de l'étiquette deviendra opérationnel pour l'objet ainsi lié (Touche ALT + lettre après le &)
- Pour supprimer un buddy on clique sur un des points rouges et on appuie sur la touche supprimer.

Layout et les spacers

1. Les Layout Vertical, Horizontal et grid : permettent de disposer les objets sur l'écran. On les dispose deux par deux
2. Les spacers permettent de gérer des espacements entre les objets
3. On utilise les layouts de layouts pour lier les layouts
4. A la fin on prend le dialogue englobant et on clique sur un layout pour attacher la fenêtre de plus haut niveau aux objets



- On sélectionne plusieurs layouts à la fois plutôt dans la fenêtre d'objets à droite
- On vérifie en agrandissant la fenêtre à l'exécution que les objets se déplacent correctement

Exercice : Application Layout

- Ordonner dans les Layout les champs précédemment utilisés

Splitter

- Les splitters sont des layout de taille modifiable c'est à dire qui permettent un agrandissement à la souris par l'utilisateur sur déplacement de la poignée séparatrice.
- On préférera les splitters sur les layout les plus externes de la fenêtre

QtDesigner plus loin

- Gestion des **QAction** : ce sont les items des menus qui lancent des actions
- Ajout de menus permet d'ajouter des menus horizontaux
- Les **ToolBox** et les **TabWidget** sont faciles à comprendre. Le designer permet d'en configurer leur nombre et leur contenu
- Les **QStackedWidget** permettent de positionner plusieurs ensembles de widgets dont un seul est rendu visible en fonction du contexte. C'est un bon moyen de préparer ces différents plans au designer et de faire varier le contenu dynamiquement
- Les **QGroupBox** regroupent visuellement des éléments de base par catégorie
- Les **QscrollArea** permettent de définir des zones de scrolling
- **Éditeur de signaux et slots :** permet de définir des correspondances entre un signal émis et une méthode appelée au déclenchement (ces méthodes s'appellent des slots)

Exercice : Découverte des slots et signaux

- Créer un nouveau projet Application graphique Qt
- Éditer le formulaire de la fenêtre principale ajouter un push

bouton
indiquant «sortir»

- dans l'onglet du bas Editeur de signal/slot faire en sorte que

lorsqu'on clique le bouton on ferme la fenêtre. Ceci se fait à nouveau de manière intuitive et sera expliqué plus tard.

- Compiler et exécuter

Slots et signaux

Slots et signaux

- Les objets Qt sont munis de Signaux qui sont déclenchés souvent par un click, la touche retour pressée, un caractère entré.
- La macro Q_OBJECT doit être présente dans le fichier .h
- l'appel du signal se fait dans le cpp par

```
connect (pointeurSurObjet1,SIGNAL(clicked()),
         PointeursurObjet2,SLOT(Action())     );
```

- Un slot est une méthode qui est déclarée dans le header de Classe derrière les mots clés *private slots:*

```
private slots:
    void Action();
```

- Le slot peut s'utiliser comme une méthode publique ou privée
- La nouvelle syntaxe C++11 du connect est la suivante

~~cpp connect (pointeurSurObjet1,&pointeurSurObjet1::clicked()), PointeursurObjet2,&PointeurSurObjet2::Action));~~

En pratique

- En général on connecte un Signal sur un objet graphique accédé par

```
ui->pushButtonQuit
```

- En général, on appellera un slot défini dans la classe courante, et on se sert du pointeur *this* qui désigne la classe courante.
- Cela donne une déclaration du connect du type :

```
connect(ui->pushButtonQuit,SIGNAL(clicked()),this,SLOT(Quit()));
```

Raccourcis Designer

- Dans le designer on peut déclarer le *connect* via la fenêtre idoine mais ce n'est pas conseillé car en pratique :
  - nous devrons gérer plusieurs dizaines de slots et de signaux de natures différentes par classe et il est préférable de disposer du source que d'une vue uniquement graphique
  - nous verrons que nous serons amenés à gérer des `disconnect` dans certains cas comme expliqué dans la suite, suivi de nouveaux `connect`
- Cette méthode doit donc être réservée à des cas d'interfaces simples avec peu de connexions

Exercice: déclaration Slot

- Reprendre le dernier exercice en supprimant l'appel au slot dans l'interface graphique
- Insérer un connect dans la classe représentant la fenêtre principale.

Signaux avec arguments

- Dans le cas d'un bouton nous avons utilisé un signal *clicked()*
- Prenons le cas d'un *QComboBox* qui gère une liste de textes. Un des signaux correspondant au *QComboBox* est `currentTextChanged(const QString & text)`, ce signal prend un argument.
- le connect associé sera

```
connect (ui->comboBox,SIGNAL(currentTextChanged(QString)),
            this,SLOT(TextChanged(QString)));
```

- Deux remarques s'imposent :
  - Nous devons fournir un slot reprenant exactement la même signature que le signal appelant à savoir *QString()*
  - ici le slot sera de la forme `void TextChanged(QString st) {..}`
  - Dans la méthode connect, on n'utilise pas les arguments mais uniquement les signatures.

Cas de disconnect

- Prenons par exemple le cas d'un *QCombobox* qui gère une liste d'items
- supposons que nous positionnons un connect de la façon suivante

```
connect ( ui->combobox,SIGNAL(currentTextChanged(QString)),
    this,SLOT(TextChanged(QString()));
```

- Imaginons que dynamiquement quelque part dans le programme nous voulions ajouter une valeur dans la Combobox. Nous ne souhaitons pas nécessairement que le slot soit appelé parce que le texte à changé.
- Dans ce cas là nous allons être amené à effectuer un :

```
ui->combobox->disconnect();
```

- Il sera suivi de la modification du *combobox* puis d'un nouveau connect

Déclaration de Signaux

- Les objets véhiculent leurs signaux
- Voir la documentation de QPushButton pour voir la liste des signaux définis pour QPushButton
- Chaque objet hérite également des signaux de ses parents
- On peut déclarer un signal dans le header de la classe comme suit :

```
signals:
    void MonSignal(type valeur);
```

Emission d'un signal

- dans le corps du programme le signal précédent sera envoyé par la syntaxe :

```
emit MonSignal(valeur);
```

- On peut choisir un signal avec ou sans paramètre. Dans le cas où l'on donne un paramètre, on oblige le slot à être déclaré avec le même paramètre
- Dans ce cas les paramètres doivent être passés au connect sous la forme suivante

```
connect ( val, SIGNAL(MonSignal(type)),
        this,SLOT(MonSlot(type)));
```

Nouvelles notation du connect en Qt5

- On peut remplacer

```
connect (ui->comboBox,SIGNAL(currentTextChanged(QString)),this,SLOT(Change(QStr
```

- par

```
connect (ui->comboBox,&QComboBox::currentTextChanged,this,&MainWindow::Change)
```

- Dans ce cas les arguments ne sont plus passés (mais ils existent), la méthode Change n'a plus besoin d'être un slot et la directive Q_OBJECT n'est plus obligatoire. Les arguments peuvent être castés entre la fonction 1 et la 2
- Par contre la méthode appelée ne doit pas être surchargée, pas plus que d'avoir un argument avec une valeur par défaut
- Il faut que les classes objet émetteur et récepteur dérivent de QObjet

Classes Qt

Rubriques

- Les types de base
- Classes QMap,QDialog, QMessageBox, QWidget, QFileDialog
- Classes QRessource, QSettings
- QRegexp

Les types de base

Les chaînes QString

- `#include <QString>`
- Manipulent toutes les chaînes de caractères

```
QString chaine = "Bonjour" ;// par transtypage
QString chaine("Bonjour");
```

- `chaine.isEmpty()` : renvoie vrai si la chaîne est vide
- `+` concatène deux chaînes de caractères

QString méthodes principales

- *split* sépare une chaine de caractères en QStringList (voir plus loin)
- De nombreuses fonctions utilisent les expressions régulières (voir plus loin)
- fonctions de conversion arg :

```
QString("Bonjour %1 %2").arg(prenom).arg(nom);
```

- conversion entier vers chaîne de caractères;

```
QString::setNum ( int n, int base = 10 ) :
QString::toInt ( bool *ok , int base =10) l'inverse
```

qDebug

- Permet l'affichage à la console d'une chaîne de caractères char* mais pas d'une QString

```
qDebug(qPrintable(QString("c'est un peu bête!"));
```

- par contre avec l'include de QDebug on peut utiliser la syntaxe

```
#include <QDebug>
int i; QString str;
qDebug()<<"Bonjour"<<i<<str;
```

- Cette syntaxe présente l'avantage de pouvoir disposer de l'affichage de la plupart des types Qt de int, à QVariant en passant par QStringList

Exercice : message console

- Créer une application graphique qui présente la saisie d'une

chaîne de caractères

- une fois la chaîne saisie, l'afficher sur la console
- Variante : afficher la chaîne de caractères en lettres majuscules
- PS : Servez vous de l'aide de QString

QList

- `#include <QList>`

```
QList <int> list ;
list <<1<<2<<3;
QList <QDate> dateList ;
```

- Instruction `foreach` (remplacée par la boucle for du C++11)

```
foreach ( int elem, list ) { qDebug()<<elem;}
for ( auto elem :list ) { qDebug()<<elem;}
```

- principales méthodes : `count()`, `takeAt()`, `pop_back()`, `pop_front()`, `push_back()`, `push_front()`,`contains()`, `isEmpty()`, `at()`
- La méthode at(i) renvoie un élément de type constante, qui ne peut pas être modifié
- Pour modifier un élément de la liste on prend l'élément avec la notation `[]`

```
list[1]=x;
```

QStringList

- QStringList est équivalent à `QList <QString>`
- avec des méthodes supplémentaires `join` (inverse de split) :

```
QString chaine=liste.join(const QString & separator) const
```

- `removeDupplicates()`, `indexOf()`
- C'est un type très utilisé dans beaucoup d'objets

Exercice : QCompleter

- QCompleter permet de compléter une zone QLineEdit par un menu d'aide

à la saisie contenant des champs prédéfinis.

- Les items de ce menu sont contenus dans une chaine QStringList
- Construire une interface contenant une ligne de saisie et introduire

une complétion avec une liste de quelques prénoms

QVariant

- C'est un type contenant des données agissant comme les unions de plusieurs types c'est à dire permettant de stocker des informations que l'on peut convertir en un type voulu.
- **Exemple :**

```
QVariant v(1000);
int x = v.toInt(); // 1000
QString y = v.toString(); // "1000"
QVariant v2("toto");
v2.toInt() // 0 car ne peut être converti
```

Classes QMap,QDialog, QMessageBox, QWidget, QFileDialog

Exercice : découverte de QDate

- Concevoir une interface qui demande la saisie d'une date;
- Afficher une zone qui demande un entier de -100 à 100
- Dès que la zone de demande de l'entier est modifiée

afficher dans une zone la date ajoutée au nombre de jours de la zone entière.

- ne pas oublier un bouton quitter
- Indices :
  - utiliser un QSpinbox pour l'entier,
  - utilisez un Label pour afficher la date calculée.

QMap et QHash

- QMap et QHash: types templates utilisant deux types génériques :
  - Key pour les clés : généralement des chaînes de caractères
  - T : pour les valeurs
- **Exemple :**

```
QMap <QString, int> map
map ["un"]=1;
map["deux"]=2;
```

- QHash plus rapide, mais pas ordonné sur les clés
- la méthode *keys* donne la liste des indices ("un", "deux")
- la méthode *values* la liste des valeurs (1,2)
- méthodes count(), contains() etc . . .

Exercice : QHash

- Initialiser un QHash avec
  - une clé prénom
  - en valeur l'âge correspondant
- Mettre le prénom dans une QCombobox
- Afficher l'âge dans un QLineEdit en lecture seule après modification de l'item courant de la combobox

QFile QDir QFileInfo

- **QFile :** gère les entrées sorties fichiers
  - **copy**, **exists**, **remove**, **rename**
  - **open** : ouverture en lecture/écriture
  - **readAll()** , **readLine()**: lecture du contenu (hérité de QIODevice)
- **QDir :** gère les répertoires (les dossiers) :
  - *mkdir*, *exists*, *remove*, *cd*, *absolutepath*
  - *homepath* retourne le répertoire de l'utilisateur courant
  - on entre les répertoires (quelle que soit la plate-forme) séparés par des caractères /
- **QFileInfo :** gère les informations sur les fichiers
  - QDir::entryInfoList renvoie une liste de QFileInfo donnant les informations propres à chaque fichier du répertoire

QDialog

- Dans le projet en cours, aller dans le menu de gauche projet, *click droit* puis *ajouter nouveau* choisir *Classe d'interface graphique Qt Designer*
- Nous constatons la création des fichiers du type :
  - MonDialogue.ui
  - MonDialogue.cpp
  - MonDialogue.h
- Quand on veut lancer le dialogue on procède comme suit dans la classe appelante

```
#include "MonDialogue.h"
MonDialogue *fenetre= new MonDialogue(this);
fenetre->exec() ;// pour un dialogue bloquant
fenetre->show() ;// pour un dialogue non bloquant
```

Exercice fenêtre A propos

- Bâtir une application comprenant
  - Une zone de plaintext
  - Un menu fichier : ouvrir, quitter, un sous menu derniers fichiers ouverts
  - Un menu aide comprenant une entrée “à propos”
  - Activer l’action quitter
  - Activer l’action à propos en affichant une fenêtre de dialogue donnant le numéro de version et le nom du programme

De l'importance du this

- Quand on ouvre une fenêtre par

```
QDialog * p = new QDialog(this)
```

- Le pointeur this indique à la fenêtre QDialog que le parent est la classe courante donc que QDialog est une fenêtre fille
- Cela a plusieurs incidences :
  - la fenêtre fille sera ouverte sous la fenêtre parente
  - le garbage collector pourra désallouer la mémoire de la classe fille si plus aucun n'objet de l'adresse. Si on ne faisait pas d'allocation dynamique la fenêtre serait détruite après le bloc courant de sa déclaration.

QMessageBox

- Permet d'afficher une fenêtre d'attention ou d'alerte

```
QMessageBox msgBox(this);
msgBox.setText(QString("On va quitter"));
msgBox.setInformativeText(QString("Êtes-vous sûr?");
msgBox.setStandardButtons(QMessageBox::Save
    |QMessageBox::Discard|QMessageBox::Cancel);
msgBox.setDefaultButton(QMessageBox::Save);
int ret = msgBox.exec();
```

QFileDialog

- Permet d'ouvrir une boite de dialogue permettant la sélection d'un fichier

```
QString fileName = QFileDialog::getOpenFileName(this,
QString("Ouvrir une image"), "/home/gilles",
        "Image Files (*.png *.jpg *.bmp)");
```

Exercice QDialog

- Bâtir une interface comprenant une zone QTextEdit
- Dans le menu proposer un sous menu Fichier, un sous Menu Aide
- Dans le menu Fichier proposer une action ouvrir fichier, quitter et derniers fichiers ouverts (remplis avec quelques noms de fichiers factices)
- Activer l'action Quitter
- Activer l'action Ouvrir fichier
- Activer une action à Propos dans le menu d'aide qui affiche une boîte de dialogue indiquant la version de l'application

Classes QRessource, QSettings

QResource

- permet de ranger des objets (images, fichiers textes, icônes, binaires) dans le corps du programme ce qui évite de gérer la gestion de ces fichiers sur le disque.
- Ces rangements suivent une hiérarchie arborescente comme un disque interne auquel on accède par le chemin ':/arborescence'
- Dans l'onglet projet (colonne gauche,) en regard du nom du projet après un click droit, ajouter **fichier Qt** / **ressource**
- Ajouter les préfixes qui sont les racines de l'arborescence puis sous chaque préfixe ajouter les fichiers
- **Attention :** la ressource est renseignée dans le fichier pro en regard du mot clé RESOURCES, donc une fois la ressource ajoutée, faire un qmake pour qu'elle soit prise en compte.

Utilisation d'une ressource

- Le fichier ressource ressemble à ceci

```
<RCC>
   <qresource prefix="/forms">
        <file>mainwindow.ui</file>
   </qresource>
</RCC>
```

- utilisation avec nom de fichier sous la forme `:/forms/mainwindow.ui`
- avec les méthodes nanipulant des noms de fichiers
- voir QRessource pour ajouter des fichiers dynamiquement

Exercice : affichage image

- Mettre une photo au format Jpeg en ressource
- en utilisant un Qlabel afficher cette photo.
- Ceci se fait uniquement avec designer

QFont

- QFont permet de définir une police de caractères.
- Une police de caractères est définie (comme le montre le constructeur avec tous les paramètres) par sa famille(ex : times), sa taille exprimé en point (PointSize), son épaisseur (weight qui va de QFont::Thin à QFont::Black) et le fait qu'elle soit en italique ou pas.
- Lorsqu'on instancie une variable QFont sans paramètre particulier elle contient la police de caractères.
- Le type QFont peut être utilisé comme paramètre d'un signal mais par contre, on ne peut faire une conversion de QVariant vers QFont.
- La méthode QFont::toString() permet de transformer la police en une chaîne de caractères contenant toutes les caractéristiques de la police.

QSettings

- permet de lire et écrire des paramètres du programme dans un fichier au format ini ou XML
- Le fichier est rangé par défaut dans un fichier/répertoire propre à chaque OS
- Mais on peut choisir le format et le nom du fichier de configuration

```
QSettings maconf ( "ma société","mon logiciel");
maconf.setValue("polices/taille",10);
QSetttings settings("ma société","mon logiciel");
int margin = settings.value("editor/wrapMargin").toInt();
```

- On peut utiliser un QSettings avec un nom de fichier Ini via le constructeur

```
QSettings s(QString fileName);
```

QSettings (suite)

- Sous Windows en format natif permet par extension de lire des clés de la table des registres
- En mode ini un fichier sous $HOME/.config/Masociété/monlogiciel.conf

```
[police]
    taille=8
```

- Les QSettings peuvent être définis globalement au niveau application pour ne plus avoir besoin d'argument :

```
QCoreApplication::setOrganizationDomain("Ma société");
QCoreApplication::setApplicationName("Mon logiciel");
```

- Ainsi on utilise

```
QSettings s;
```

- La méthode static QSettings::setDefaultFormat(Format format) permet de définir globalement le format du fichier de setting

Exercice : QSettings QDialog et QMenu

- Écrire un programme qui affiche dans son menu une option dialogue de configuration qui permet de sélectionner la police de caractères
- Au prochain lancement, l'affichage de la police sauvée doit être conservée
- On pourra faire en sorte que la validation du changement de police impacte la fenêtre mère.

QRegexp

Présentation

- Appelées en anglais `regular expression` (souvent abrégé en `regexp`), ce qui a donné en français les deux traductions `expressions rationnelles` et `expressions régulières`
- Les expressions régulières permettent de retrouver un motif dans un fichier texte ou dans des chaînes de caractères
- **Par exemple :** ^a correspondra à toutes les lignes commençant par a
- Elles permettent également de faire des subtitutions de motifs
- On les retrouve dans beaucoup de fonctions de recherche sous Unix, dans les librairies comme Qt, dans les éditeurs de texte qui permettent ce genre de recherche poussée
- C'est un bon moyen de s'initier que d'utiliser un éditeur qui possède cette option

Caractères

- . n'importe quel caractère sauf \n
- [abc] a b ou c
- [a-z] de a à z
- [^cd] ni c ni d
- ^x commençant par x
- x$ finissant par x
- ? le caractère précédent 0 ou 1 fois
- * le caractère précédent 0 ou n fois
- + le caractère précédent 1 ou n fois
- {n} le caractère précédent n fois
- {,p} le caractère précédent au plus p fois
- {n,p} le caractère précédent au moins n fois et au plus p fois
- \ rend le rôle usuel aux caractères \ | ( ) [ ] { } ^ $ * + ? .
- | choix entre plusieurs chaînes de caractères

Premiers exemples

- `chat|chien` correspond aux chaînes de caractères chat, chien ou chaton
- `[cC]hat|[cC]hien` correspond aux chaînes chat,Chat, chien ou Chien
- `chu+t` correspond à chut,chuut,chuuut etc..
- `a[ou]+` correspond à aou, ao, auuu, aououuuoou, etc. . .
- `peu[xt]?` correspond à peu,peux et peut (et seulement à ces chaînes)
- `^[st]ac` représente les chaînes sac et tac en début de ligne.
- `[st]ac$` représente les chaînes sac et tac en fin de ligne ou de texte
- `^trax$` représente la chaîne trax seule sur une ligne.

Autres exemples

- Pour l'expression `[a-zA-Z]+= *[0-9]+ *;` la ligne
  - Toto= 2; : correspond
  - Toto2 =3 ; : ne correspond pas
- Pour l'expression `[a-zA-Z]+ *[+*/-]{1}`
  - char * : convient
  - toto +23 : convient
  - toto++3 : convient aussi
  - On remarque que le - de l'expression régulière est à la dernière place pour ne pas prendre la signification intervalle, on aurait également pu le mettre en première place.

Ensembles

- `\d` : un chiffre, équivalent à `[0-9]` (d comme digit)
- `\D` : un non-numérique, équivalent à `[^0-9]`
- `\w` : un alphanumérique, équivalent à `[0-9a-zA-Z_]` (w comme word)
- `\W` : un non-alphanumérique, équivalent à `[^0-9a-zA-Z_]`
- `\s` : un espacement, équivalent à `[ \n\t\r\f]` (s comme space)
- `\S` : un non-espacement, équivalent à `[^ \n\t\r\f]`
- `\b` : marque de début ou de fin de mot

Captures

- L'emploi de parenthèses dans une chaîne permet de capturer les valeurs entre parenthèses, avec une syntaxe particulière à chaque environnement
- Néanmoins les valeurs capturées sont toujours accessibles par leur numéro d'apparition dans les parenthèses.
- Ainsi `([a-zA-Z]+) *=*([0-9]+) *;` permettra de récupérer les valeurs
  - en shell \1 \2
  - en Qt cap(1) cap(2)

QRegExp : expressions régulières

```
QRegExp expression("(\\d+)");
```

- Doubler les \
- Utilisées dans des fonctions sur les Qstring (replace, contains etc)

```
expression.indexIn("ok1232") ;
QString captured = expression.cap(1); // captured 1232
```

Exercice : Expressions régulières

- Afficher les fichiers de votre répertoire d'accueil dont l'extension comprend 3 caractères alphanumériques

Classes évoluées

Rubriques

- Bases de données
- Modèle Vue Controleur
- Plus loin avec les MVC
- QNetwork

Bases de données

Bases de données compatibles Qt

- **SQLite :** base de données SQLite3 employée dans le domaine de l'embarqué. Un seul fichier, facile à manipuler. Présente par défaut sur MacOSX et Android - Driver QSQLITE (ou QSQLITE2 pour la version 2)
- **Mysql :** la base de données la plus employée dans le domaine des sites Web (Driver QMYSQL)
- **Oracle :** la plus complète et la plus onéreuse en terme de licence et de mise en oeuvre - Driver QOCI
- **Microsoft SQL Server :** ne tourne que sous Windows - Driver QODBC valable pour d'autres Bases ODBC
- **PostgreSQL :** plus ou moins équivalente à MySql - Driver QPSQL
- **IBM DB2 :** DB2 d'IBM
- **Interbase :** de Borland - Driver QIBASE

Intérêt

- Optimisé pour l'interrogation de gros volumes d'informations
- Stockage des données structurées
- Language universel SQL indépendant des bases de données
- Rapidité d'accès
- Possibilité d'ajout d'informations sans changer les requêtes SQL
- Outils de maintenance, élimination de doublons, effacement des données obsolètes

Notions de lanqage SQL

- Création d'une table

```
CREATE TABLE ages ( nom TEXT, age INTEGER)
```

- Insertion dans une table

```
INSERT INTO ages ( nom, age) VALUES ('Dupont',43)
```

- Sélections diverses dans une table

```
SELECT nom FROM ages WHERE age > 3
SELECT * FROM ages WHERE age = 2
SELECT nom,age FROM ages WHERE nom = 'Dupont'
```

QSqlDatabase

- Ouverture base de données une seule fois dans l'ensemble du projet (dumoins si on a une seule base)

```
QSqlDatabase db = QSqlDatabase::addDatabase("QSQLITE");
db.setDatabaseName("mabase.db");
bool ok = db.open();
```

- **Attention :** ajouter `QT += sql` dans le fichier pro
- Vérifier que Sqlite3 ou que Sqliteman est installé si on veut débuguer les requêtes sqlite

SqlQuery

- Ordres d'insertion :

```
QSqlQuery query("INSERT INTO person (id, forename, surname) VALUES( 1,'Marcel',
```

ou

```
QString requete="INSERT INTO person (id, forename, surname) VALUES('2','Marcel'
QSqlQuery query2 ;
query2.exec(requete);
```

- Ordres SELECT avec balayage

```
QSqlQuery query("SELECT ville FROM pays");
 while (query.next()) {
   QString ville = query.value(0).toString();
}
```

Facilités QSQLQuery

- **QSQLQuery::lastQuery() :** affiche la dernière requête effectuée
- **QSQLQuery::lastError() :** donne l'erreur sur la requête.
- SQLITE est très lent en INSERT ne pas hésiter à faire une transaction par :

```
db.transaction()
QSqlQuery query("INSERT INTO person (id, forename, surname) VALUES ...;
QSqlQuery query("INSERT INTO person (id, forename, surname) VALUES ...;
db.commit();
```

- Ces deux instructions doivent encadrer l'ensemble des INSERT

Exercice Insert Base de données

- Récupérer la table comsimp des communes de France sur le site de l'INSEE
- En faire une table dans une base de données SQLLite

Modèle Vue Controleur

Modèle MVC QSqlModel

- On définit un modèle

```
QSqlTableModel *model = new QSqlTableModel();
```

- On l'initialise :

```
model->setTable("ville");
model->select();
```

- On associe un QtableView à ce modèle

```
ui->tableWiew->setModel(model);
```

- QTableView est un objet de visualisation disponible dans le Designer.
- Ainsi : on place l'objet TableView dans l'interface, on lui applique un modèle que l'on a initialisé

Récupération des informations du modèle

- On ne récupère pas les informations de zone cliquée de QTableView mais du modèle par exemple par un signal

```
connect (ui->tableView,SIGNAL(pressed(QModelIndex)),
  this, SLOT (AfficheInfo(QModelIndex)));

void AfficheInfo( QModelIndex ind)
{
 QString val;
 val=ind.data().toString();
 //val=ind.sibling(ind.row(),ind.column()).data().toString();
 // syntaxe si on souhaite cibler sur une autre ligne colonne
}
```

Exercice SQLModel

- Écrire une application affichant la table Comsimp
- Afficher dans un champ QLineEdit la valeur de la case cliquée.

Les Views

- Le principe est toujours de laisser l'affichage s'effectuer globalement sans se préoccuper de chaque ligne d'affichage, c'est un mode rapide.
  - **QListView :** gère l'affichage des listes avec modèle
  - **QTableView :** gère l'affichage des tables avec modèle
  - **QTreeView :** gère l'affichage des arbres avec modèle
- Ces view sont associées à des modèles : QStringListModel, QSqlQueryModel, QFilesSystemModel, QSqlTableModel, QSqlRelationalTableModel
- Exemple :

```
QStringListModel *model = new QStringListModel(this);
model->setStringList(list);
ListView->setModel(model)
```

QSortFilterProxyModel

- QSortFilterProxyModel est un des mécanisme héritant de QAbstractProxyModel permettant de filtrer et de trier des données d'un modèle.
- Si un modèle a été passé à un objet comme suit :

```
QSqlQueryModel *model = new QSqlQueryModel(this);
model.setQuery(requete);
ui->tableView->setModel(model);
```

- On le déclare :

```
QSortFilterProxyModel *proxy = new QSortFilterProxyModel(this);

QSqlQueryModel *source = new QSqlQueryModel(this);
proxy->setSourceModel(source);
ui->tableView->setModel(proxy);
```

Exercice : Liste ordonnée ListView

- Afficher une liste de 10 noms dans un QListView piloté par un QStringList
- Lorsqu'on clique sur un nom il passe en dernière position de la liste

Les équivalents élément

- On peut gérer des affichages QTableWidget, QListWidget, QTreeWidget, QListWidget
- Dans chacun de ces modes, on effectue les affichages ligne et/ou colonne par ligne et/ou colonne
- L'affichage est plus lent est doit être réservé à quelques dizaines d'éléments.
- Attention avant d'implémenter une telle fonction :
  - faire un setRowNumber ou set ColumnNumber
  - on ajoute les éléments item par item via insertRow() etc. . .

Exercice : Liste ordonnée ListWidget

- Afficher la même liste via un QListWidget
- Lorsqu'on clique sur un nom il passe en dernière position de la liste

Plus loin avec les MVC

Quelques points à savoir sur le modèle vue contrôleur

- Pour modifier la taille des colonnes on utilise la méthode (en général sur l'évènement resizeEvent(QResizeEvent *e) qui est étudié plus loin

```
tableView->setColumnWidth(int i, int taille);
```

- Pour afficher des noms de colonne à un tableView on utilise

```
model->setHeaderData(1,Qt::Horizontal,tr("Category"));
```

- On peut rendre la selection unitaire ou multi sélection par :

```
tableview->setSelectionMode(QAbstractItemView::SingleSelection);
```

- On peut forcer la sélection à une ligne entière par

```
tableview->setSelectionBehavior(QAbstractItemView::SelectRows);
```

Construire son propre modèle

- On peut tout à fait construire son propre modèle et nous prenons l'exemple d'un modèle construit pour un QTableView
- Pour créer le modèle on dérive la classe QAbstractTableModel

```
class MonModel : public QAbstractTableModel
{
public:
    MonModel(QObject *parent=0);
{
```

- En compilant la classe dérivée, un message d'erreur nous indiquera que la classe QAbstractTableModel est abstraite et qu'il faut redéfinir trois méthodes virtuelles pures :

```
int rowCount(const QModelIndex &parent) const;
int columnCount(const QModelIndex &parent) const;
QVariant data(const QModelIndex &index, int role) const;
```

Le paramètre role

- Le paramètre rôle peut prendre un certain nombre de valeurs : Qt::DisplayRole,DecorationRole etc..
- C'est souvent le role DisplayRole qui nous intéresse :
- Exemple

```
QVariant VCardModel::data(const QModelIndex &index, int role) const
{
    if (role == Qt::DisplayRole)
        {
            return m_vcard->getValue(index.row(),index.column());
        }
    return QVariant();
}
```

Les delegates

- Le composant delegate permet à partir d'une vue
  - de personnaliser l'édition des éléments au moyen d'un editor ;
  - de personnaliser le rendu des éléments à l'intérieur d'une vue.
- Par défaut les vues gèrent une édition éventuelle et un rendu par défaut car elles disposent d'un delegate par défaut : QItemDelegate et QStyleItemDelegate
- La classe pour définir un nouveau delegate est QAbstractItemDelegate
- Si on veut modifier le delegate il faut définir

```
void QAbstractItemView::setItemDelegate ( QAbstractItemDelegate * delegate )
```

QNetwork

QNetworkAccessManager

- La classe QNetworkAccessManager permet d'utiliser le signal finished() pour connecter un slot de traitement de la réception.
- Elle nécessite d'ajouter dans le fichier pro le module network
- L'url appelée est précisée par la méthode get :

```
QNetworkAccessManager *manager =
                                        new QNetworkAccessManager(this);
 connect(manager, SIGNAL(finished(QNetworkReply*)),
         this, SLOT(replyFinished(QNetworkReply*)));
 manager->get(QNetworkRequest(QUrl("http://qt-project.org")));
```

Exemple de traitement d'un fichier reçu

- Le slot contiendra un code analogue à ceci :

```
QFile file("Fichier.data");
file.open(QIODevice::WriteOnly);
QByteArray ba;
qint64 maxlen=2048;
bool end=false;
while (!end)
{
    ba=reply->read(maxlen);
    file.write(ba);
    if ( ba.isEmpty()) end=true;
}
file.close();
```

- On peut ajouter un signal informant de la progression dans la boucle while

Suivre les liens

- Pour que la requête suive les liens et que votre fichier de contienne pas d'information :

```
Adresse moved to :
```

- On devra utiliser la méthode suivante (depuis Q t5.6)

```
QNetworkRequest request(url);
    request.setAttribute(QNetworkRequest::FollowRedirectsAttribute,true);
    m_reply=m_network->get(request);
```

Debug

Rubriques

- Présentation
- Point d'arrêt
- Debuguer via un core dump
- Debuguer à distance

Présentation

Débugage

- Il faut compiler en mode debug via l'icône du bas *Debug/Release* qui détermine le type de compilation
- Le débugage consiste à mettre des points d'arrêt et à suivre le programme en mode pas à pas, continue, entrer dans etc. . .

Méthodes de débugage

- Quand un programme plante deux cas se présentent :
  - plantage de façon aléatoire
  - plantage à un endroit précis
- Quand c'est de façon aléatoire, il faut donner une version du programme compilé à l'utilisateur avec les options de débugages.
  - Si on est sous Unix on règle la génération des fichiers core par ulimit -c unlimited
  - Ensuite on lance le débugueur l'exécutable auquel on ajoute en deuxième argument le fichier core
- Dans le cas où le plantage est systématique on lance le débugage pas à pas

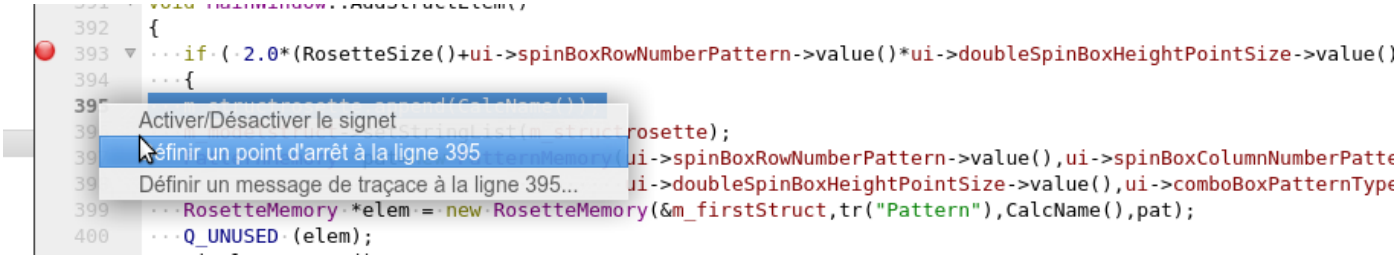
Débugage pas à pas

- Lorsqu'un programme plante, un point d'arrêt permet d'arrêter le programme à un point donné au début du programme
- En cas de problème on peut afficher les log de débugage par le menu *Fenêtre / Vues / Journal de débogage*

Poser un point d'arrêt

- Dans l'éditeur, au niveau des numéros de ligne le bouton droit de la

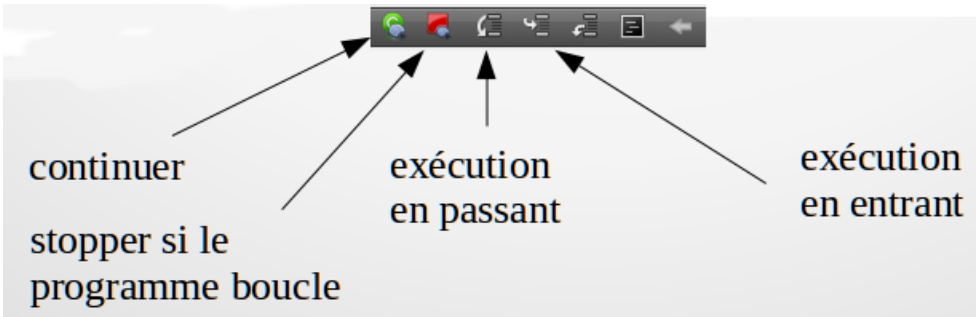
souris actionner le menu proposant de poser un point d'arrêt



- Un point d'arrêt posé est matérialisé par un point rouge comme montré un peu plus haut - Un point d'arrêt peut être désactivé par le même menu - Ensuite cliquer sur l'icone de debug 

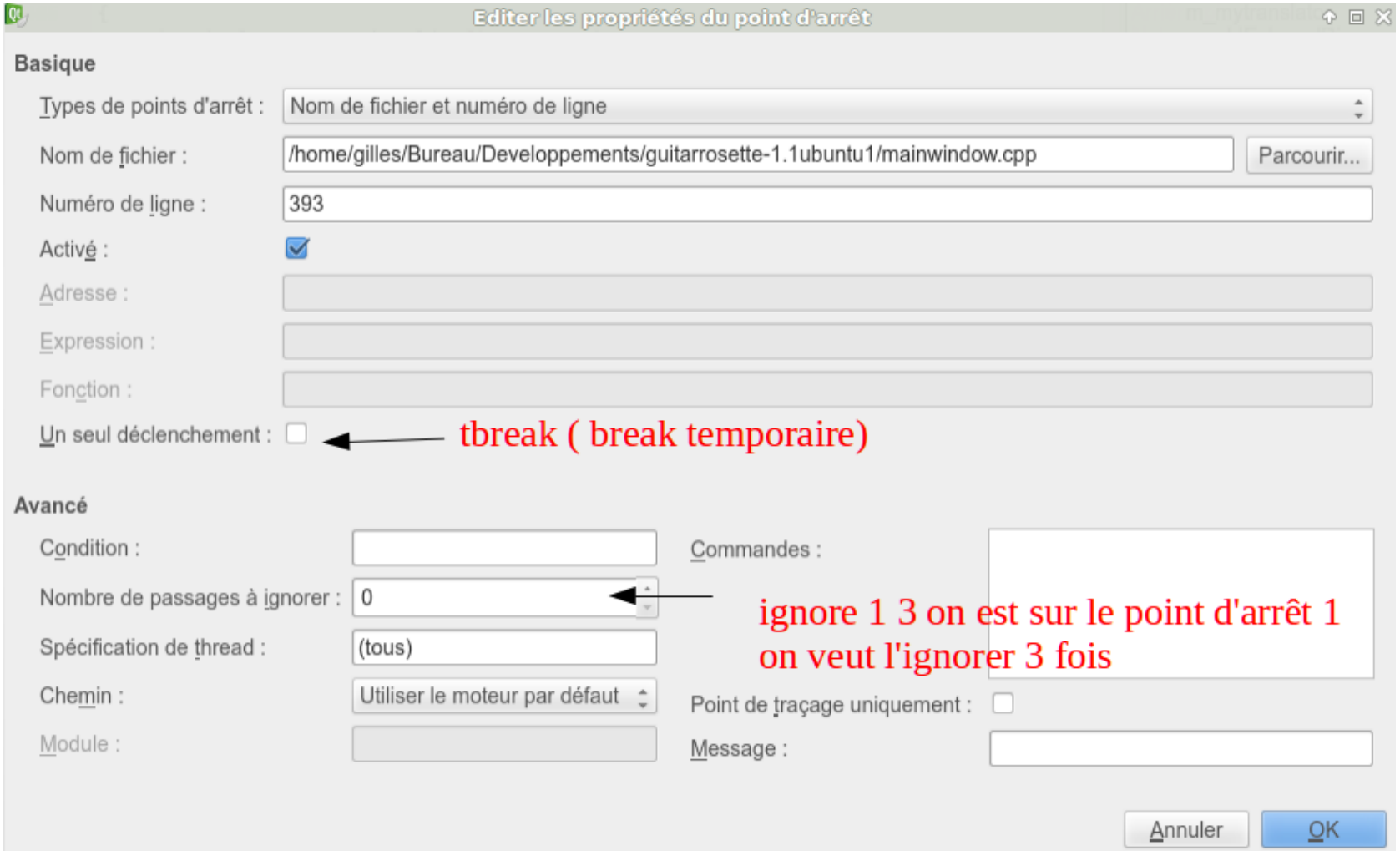
Inspecter le programme

- Une fois le programme lancé il se bloque s'il passe par le point d'arrêt posé
- Vous pouvez explorer les variables.
- Vérifier que l'item Fenêtres / Vues / Locales et expression est cochée vous verrez alors apparaître les variables sur la partie droite de l'écran (on peut modifier la valeur des variables locales en cliquant dessus)
- on peut exécuter le programme pas à pas par la barre



Point d'arrêt

Point d'arrêt : plus loin



gdb : point d'arrêt

- On place des points d'arrêt par les commandes
  - **break** (b) nom de fonction (fonctionne sans table des symboles)
  - **break TestClass::testfunc()** : point d'arrêt dans une méthode d'une classe
  - **break (b) numéro de ligne** de programme (nécessite la table des symboles)
- on peut ajouter if condition après un break (condition utilise les variables du programme). Ceci permet de s'arrêter à un point donné que si une condition est remplie :

```
b 12 if ( compteur == 2 )
```

- La condition peut être ajoutée sur un point d'arrêt existant par la syntaxe

```
condition numerobreak expression
```

- Pour supprimer une condition `condition numerobreak`
- *numerobreak* n'est pas la ligne du programme mais le numéro du break donné par la commande `info break` qui liste les points d'arrêt.
- on peut aussi demander à passer sur un point d'arrêt un certain nombre de fois avant de s'arrêter par

```
ignore numbreak compteur
```

Break avec liste de commandes

- les watchdogs, les breakpoints et les catchpoints peuvent être assignés à des suites de commandes
- les commandes sont encadrées par les mots clés `commands` et `end`
- à l'intérieur on utilise généralement le mot clé `silent` qui indique au break de rester en mode silencieux on inclue un ordre continue, step ou next, si on souhaite que le point d'arrêt se comporte comme un point de modification de code et non plus comme un point d'arrêt

```
break 567
commands
silent
set x = y + 5
continue
end
```

- Pour détruire une liste de commandes on déclare une séquence vide entre `commands` et `end`

Autres points d'arrêts

- la vue des expressions et des variables locales permet de visualiser une expression
- si on clique via le bouton droit sur une expression dans cette vue, peut poser un point d'arrêt via Add data breackpoint.
- Ce point d'arrêt sera alors effectif lorsque que la variable sera modifiée

Debuguer via un core dump

Debug après plantage : core dump

- Sous Linux on peut facilement mettre en place un mécanisme permettant de débuguer un programme en analysant après coup un fichier trace appelé *core*.
- C'est via le débugueur que ce fichier va être rejoué.
- Il faut cependant s'assurer que ce fichier est généré en cas de plantage, ce qui n'est pas forcément le cas sur toutes les plate-formes.
- Il faut également connaître précisément la version du logiciel qui a planté et être en mesure de disposer des mêmes sources. Ceci est rendu possible par l'utilisation de gestionnaires de versions dont le plus répandu est aujourd'hui *git*.
- Il faut donc savoir précisément actionner la génération d'un fichier core (core dump)
- Il est conseillé de renseigner la version git à l'intérieur de l'exécutable afin de savoir la retrouver via l'exploration d'une variable par exemple.
- En outre, il faut garder à l'esprit que beaucoup de distributions génèrent ce fichier core mais laissent le programme Apport l'intercepter, en demandant l'autorisation à l'utilisateur d'envoyer le core au développeur. Il nous faut donc savoir désactiver ce mécanisme

Tuning de génération du core

- Vérification qu'un fichier core n'est pas intercepté

```
cat /proc/sys/kernel/core_pattern
```

- par exemple par `|/usr/share/apport/apport %p %s %c`
- que le fichier /etc/security/limits.conf contienne * soft core unlimited et * hard core unlimited
- Si on veut que la modification soit temporaire, on exécute

```
- en root : echo /tmp/core.prog_%e.sig_%s.proc_%p>/proc/sys/kernel/core_pattern
- en utilisateur : ulimit -c unlimited
```

- Si on veut qu'elle soit permanente :
  - ulimit se règle sous Debian en mettant dans le fichier `/etc/security/limits.conf` : `* soft core unlimited`
  - ajouter `kernel.core_pattern = /tmp/core.prog_%e.sig_%s.proc_%p` dans un fichier `/etc/sysctl.d/50-coredump.conf`
  - supprimer le programme apport (apt-get remove apport)
- On peut vérifier que le répertoire /tmp contient bien un fichier core après lancement de : `sleep 100 & kill -SIGSEGV %%` (%% est une variable bash représentant le PID de la dernière tâche suspendue)

Debug Core Qt

- On s'assure que le programme qui a produit le core était bien compilé en mode Debug
- Dans Qt Creator on lance Déboguer/Commancer le débogage/Charger un fichier core
- le programme doit être généré avec les options de debug
- le fichier core à charger doit être de la forme `/tmp/core.programm_bash.sig_11.proc_29255`
- Si on est dans le shadow on cherche le programme dans le répertoire build, sinon QtCreator le trouve seul
- Le débugueur montre l'instruction qui a provoqué le plantage
- Si on est en assembleur ou dans une librairie on remonte via autant de commandes *up* que nécessaire jusqu'à trouver son code
- on inspecte sous le débugueur les variables afin de comprendre pourquoi l'instruction a provoqué le plantage

Debuguer à distance

Debug croisé

- On appelle debug à distance la possibilité de débuguer un logiciel tournant sur un équipement distant depuis un autre équipement.
- Ce cas est très utile en embarqué où les machines cibles sont pourvues de peu d'espace de stockage et munis de processeurs de petite capacité demandant des compilations croisées. C'est pourquoi cette technique s'appelle également debug croisé.
- *gdb* permet le débug croisé via un port série ou via TCP/IP
- *gdb* permet de débuguer un code distant sur un processeur différent du processeur de la machine possédant le débugueur. Il gère même les cas de byte ordering différent.
- Sur la carte distante, on rencontre deux cas :
  - un serveur gdb est installé car on est sur une machine unixlike, on communique de débugueur client à débugueur serveur
  - un serveur n'est pas installé dans ce cas on doit implémenter sur le logiciel de la carte le protocole stub de communication de gdb.
- Nous nous bornerons aux cas d'utilisation avec le protocole gdbserver en mode TCP/IP

Installation sur l'équipement distant

- nous supposons que ce serveur a l'adresse 192.168.0.237
- sur le serveur distant installer gdbserver
  - sur architecture debian : `apt-get install gdbserver`
  - sur embarqué busybox disponible dans les config
- La machine distante et la machine host doivent disposer d'un binaire construit à partir des mêmes sources avec les librairies idoines
- sur la machine distante : le binaire compilé sans option -g, stripé ou non stripé, les sources ne sont pas nécessaires
- Sur le serveur distant on lance le serveur gdbserver avec la syntaxe suivante :

```
gdbserver 192.168.0.32:2345 programme arguments
```

- 192.168.0.32 est l'adresse IP de la machine Host
- 2345 est le port TCP que vous choisissez
- programme est le programme que vous débuguez juste compilé

IGNU

Mise en route sur l'équipement host

- Nous supposons que cet équipement a l'adresse 192.168.0.32
- Nous compilons le programme avec l'option -g
- Sur le poste de travail du débugueur on lance

```
gdb nom du programme
```

- On peut ainsi débuguer à distance après avoir informé le débugueur par la première commande :

```
target remote 192.168.0.237:2345
```

- le programme étant lancé on fait généralement un **break main** ou

un **continue**

Débug croisé à partir de QtCreator

- Dans le menu Deboguer/Commencer le débogage/Attach to running debug server
  - on doit alors entrer le port du serveur,
  - son adresse Ip
  - l'emplacement du programme local
- Lorsqu'on s'attache au serveur le programme est stoppé et on doit demander de continuer après avoir posé d'éventuels breaks
- Dans ce mode le client discute avec le serveur et on peut suivre pas à pas le déroulement du programme

Images

Rubriques

- Exemple éditeur de texte
- QPainter
- Les images
- GraphicsScene

Exemple éditeur de texte

Exemple de dérivation de classe

- Dans QPlainTextEdit, le bouton droit de la souris dans l'éditeur provoque un menu
- Nous voulons ajouter un item dans ce menu
- Nous constatons que la méthode suivante est *protected virtuelle*

```
void QPlainTextEdit::contextMenuEvent (ContextMenuEvent * event ) [virtual prot
```

- Nous construisons donc une classe *MyQPlainTextEdit* dérivée de *QPlainTextEdit* où nous implémentons la fonction `contextMenuEvent` (voir page suivante)
- Pour positionner cette classe à la place d'un objet QPlaintextEdit dans la partie Designer, on clique avec le bouton droit sur l'objet on le promeut en MyQPlainTextEdit
- On prend garde de définir un constructeur dans MyQPlainTextEdit prenant un QWidget *parent en argument

Mise en oeuvre

- Dans le .h

```
struct     MyPlainTextEdit  : public QPlainTextEdit
{
 Q_OBJECT
       MyPlainTextEdit  (QWidget *parent=0);
        void contextMenuEvent(QContextMenuEvent   *event);
private slots:
    void ActionMenu(bool);
};
```

- Dans le cpp

```
void MyPlainTextEdit::contextMenuEvent(QContextMenuEvent *ev)
{
     QMenu *menu = createStandardContextMenu();
    QAction *action = new QAction("Mon action",this);
    menu->addAction(action);
    connect ( action,SIGNAL(triggered(bool)), this,SLOT(ActionMenu(bool)));
    menu->exec(ev->globalPos());
}
```

Exercice : éditeur de texte

- Développer un éditeur de texte qui dispose d'un menu contextuel dans lequel on ajoute une entrée qui envoie un message à la console

Évènements

- Les évènements peuvent être reçus et pris en charge par n'importe quelle instance d'une sous-classe de QObject.
- Quand un évènement a lieu, Qt crée un objet d'évènement pour le représenter en construisant une instance de la sous-classe appropriée de QEvent et la délivre à une instance particulière de QObject
- On trouve les classes suivantes qui dérivent de QEvent `QResizeEvent`, `QPaintEvent`, `QMouseEvent`, `QKeyEvent`, et `QCloseEvent`
- La classe QWidget a plusieurs fonctions virtuelles protégées qui peuvent donc être utilisées dans toutes les classes dérivées de QWidget

```
virtual void    mouseDoubleClickEvent(QMouseEvent * event)
virtual void    mouseMoveEvent(QMouseEvent * event)
virtual void    mousePressEvent(QMouseEvent * event)
virtual void    mouseReleaseEvent(QMouseEvent * event)
```

Définitions

- Les événements sont déclarés comme des fonctions protégées et virtuelles
  - elles ne sont utilisables que dans des classes dérivées (protégées)
  - elles sont virtuelles, ce qui fait qu'un pointeur sur la classe mère donnera la valeur dans la classe fille.
- Ces événements sont déclenchés par une action qui peut être moins immédiate que l'envoi d'un signal parce que dans certains cas ils sont cadencés par une horloge.

QPaintEvent

- Cet événement est appelé à chaque affichage partiel ou total d'une widget qu'il soit provoqué par :
  - un ordre `repaint()` ou `update()` que l'on utilise pour demander un rafraîchissement
  - le widget a été masqué par un autre objet et il est rafraîchi
  - la fenêtre mère a été agrandie, déplacée etc. . .
- On remarque également que c'est dans cette fonction qu'on connaît la taille de l'image une fois que les objets ont été placés par des Layout

```
void maWidget::paintEvent( QPaintEvent *event )
{
      qDebug()<<this->width()<<this->height();
}
```

Événements souris

- ces événements sont appelés sur des actions de la souris

```
void maWidget::mousePressEvent(QMouseEvent *event)
{  if ( event->pos().y() > this->height()-30 )
        {  //on est en bas de l'écran    }
}

void maWidget::mouseDoubleClickEvent(QMouseEvent *event)
{
 emit DoubleClicked();
}

void maWidget::mouseMoveEvent(QMouseEvent *event)
{qDebug()<<event->pos().x();}
```

Attention : pour activer le mouseMoveEvent sans que le bouton de la souris soit appuyé faire

```
ui->centralWidget->setMouseTracking(true);
this->setMouseTracking(true);
```

Événement clavier

- Appelé à l'appui ou au relâchement d'une touche du clavier

```
virtual void    maClasse::keyPressEvent(QKeyEvent * event)
{
    if (event->key() == Qt::Key_Escape)
    {
        this->close();
    }
   if ( event->modifiers()==Qt::ControlModifier &&
    event->key()==Qt::Key_Q) {}
}

virtual void    maCLasse::keyReleaseEvent(QKeyEvent * event)
{
}
```

Exercice Double click

- Intercepter le double click en affichant un message
- Intercepter le mousePressEvent et afficher un message si

l'évènement est un click droit de la souris

QPainter

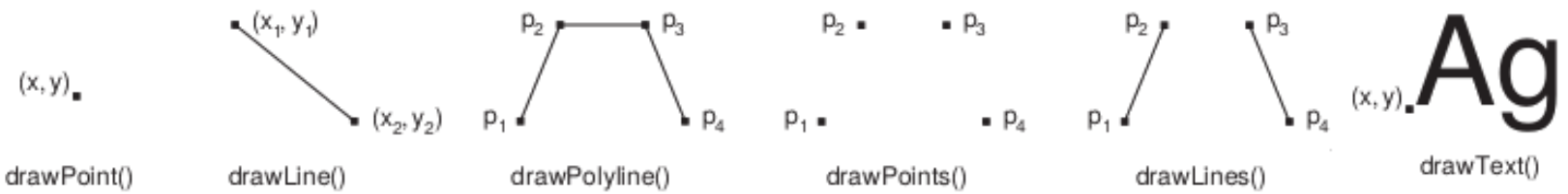
Présentation QPainter

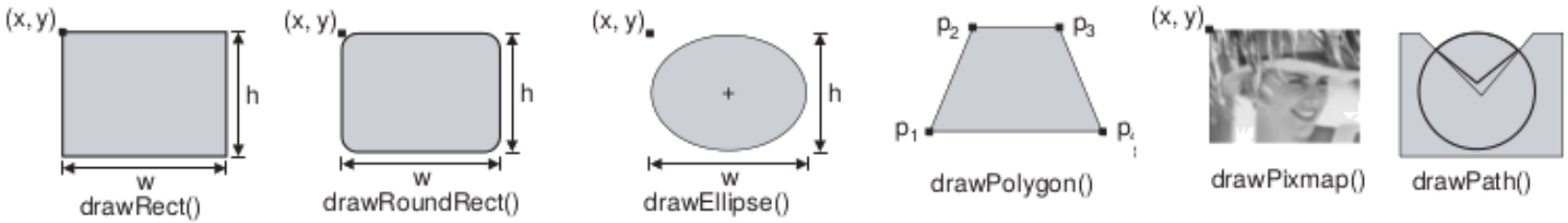
- `QPainter` est la classe de base pour dessiner au milieu de différents `QWidget`.
- La méthode `void paintEvent (QPaintEvent *e)` de la classe QWidget est une méthode virtuelle protégée. C'est elle qu'on utilise dans les classes dérivées pour bénéficier d'un `QPainter`

```
void Classe::paintEvent(QPaintEvent *e)
{
 QPainter painter(this);
 painter.drawLine(0,0,200,200);
ClasseMere::paintEvent(e) ;
}
```

Attention si la widget contient une SrollArea le QPainter painter(this) doit être remplacé par QPainter(viewport());

Les directives graphiques de QPainter

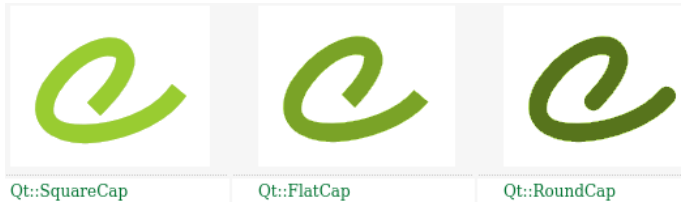




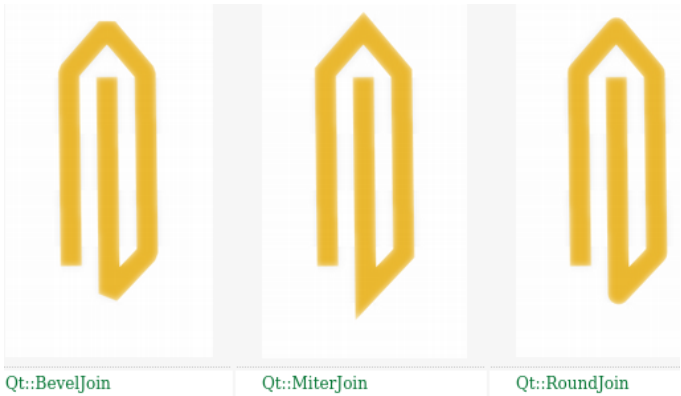
Crayon (QPen)

- Le crayon est utilisé pour tracer des lignes et les contours des formes.
- Il est constitué d'une couleur (QColor), d'une épaisseur et :
  - d'un style de trait (QPenStyle)

  - de capuchon (QPenCap)

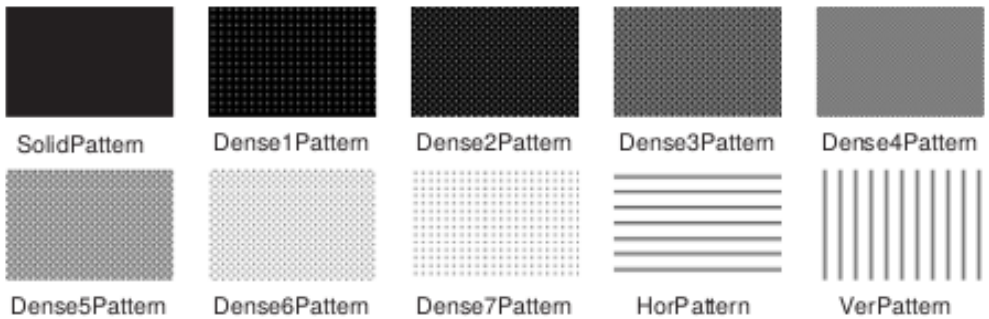


  - de jointure (QPenJoin)

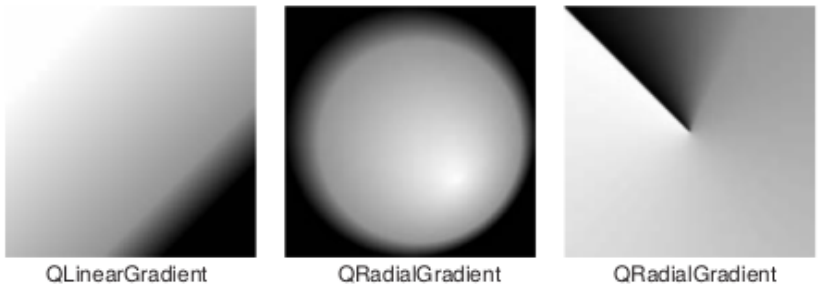


Pinceau (QBrush)

- Le pinceau permet de remplir des formes géométriques.
- Il est composé normalement d'une couleur (QColor)



- d'une texture prédéfinie
- ou d'une texture (QPattern) donnée par une image `QPixmap` répétée



- ou d'un dégradé (QGradient)

Police (QFont) et Coordonnées

- Police
  - La police est utilisée pour dessiner le texte.
  - Elle possède de nombreux attributs, dont la famille, la taille, le type d'espacement inter caractères, une capitalisation éventuelle, le graissage
  - La police courante est donnée par le constructeur QFont() sans argument
- Cordonnées
  - Par défaut le point (0, 0) se situe dans le coin supérieur gauche
  - Les coordonnées x augmentent vers la droite et les coordonnées y sont orientées vers le bas
  - Le centre d'un pixel se trouve aux coordonnées d'un "demi" pixel sauf si l'anticrénelage est activé dans quel cas les pixels avoisinant en gris clair pour donner une impression d'exactitude

Opération de transformation

- Bien qu'on puisse passer par des matrices de transformation plus ou moins complexes, on peut utiliser les fonctions simplifiées suivantes :
  - painter.translate(-10.0, -20.0);
  - painter.rotate(45.0);
  - painter.translate(+10.0, +20.0);
  - painter.drawText(rect, Qt::AlignCenter, tr("Revenue"));
- Ces actions peuvent être répétées par l'emploi d'un `QTimer` utilisé comme suit

```
updateTimer = new QTimer(this);
connect(updateTimer, SIGNAL(timeout()), this, SLOT(update()));
updateTimer->start();
```

Les images

Généralité

- `QPainter` : permet de dessiner un arc de cercle (drawArc), des coordonnées (drawChord), des lignes (drawlines), des rectangles (drawrect) des polynomes (drawConvexPolynom), des cercles ...
- La classe QPaintDevice est la classe des objets qui peuvent être dessinés par les méthodes de la classe QPainter.
- Quatre classes dérivent de QPaintDevice et reçoivent les méthodes QPainter :
  - QImage
  - QPixmap
  - QBitmap
  - QPicture

Les classes Images

- `QImage` permet d'accéder aux pixels et de traiter des Images. Rapide à dessiner, lent à afficher.
- `QPixmap` permet les manipulations et les traitement d'images destinées à être affichées dans l'application. Lent à dessiner, rapide à afficher. Utile aux motifs au répétitions.
- `QBitmap` hérite de `QPixmap` avec une profondeur de 1 bit, c'est à dire que la valeur 1 ou 0 ne permet de dessiner qu'en noir et blanc.
- `QPicture` ne se soucie pas de la résolution ou plutôt fonctionne avec la résolution courante. Le nombre d'instruction est plus limité que celui des QPixmap et QImage. Le format de QPicture est propriétaire, les primitives load et save ne savent pas lire de fichier gif ou png. C'est un mode rapide de traitement sans compression.
- `QIcon` permet d'utiliser un fichier comme une image dans les objets qui ont besoin d'afficher une icône (QPushButton,etc..)

Conversions d'un type à l'autre

- QImage vers QPixmap
  - QPixmap::convertFromImage(const QImage & image)
- QPixmap vers QBitmap (passage en noir et blanc)
  - par constructeur `QBitmap(const QPixmap & pixmap)`
- QPixmap vers Icon par constructeur : `QIcon(QPixmap pix)`
- QPainter depuis un type QImage, QBitmap, QPixmap

```
QPixmap image("fichier.png");
QPainter painter;
painter.begin(&image);
painter.drawText(...);
painter.end();
```

Lectures écritures aux formats d'image

Format	Type
BMP	Windows Bitmap
GIF	Graphic Interchange
JPG JPEG	Joint Photographic Group
PNG	Portable Network Graphics
PBM	Portable Bitmap
PGM	Portable Graymap
PPM	Portable Pixmap
XBM	X11 Bitmap
XPM	X11 Pixmap

Exemple de création de fichiers Image

```
// Création du pixmap
QPixmap *pixmap = new QPixmap(30,30);
pixmap->fill(Qt::transparent);
QPainter *painter= new QPainter(pixmap);
// Définition du stylo et de la brosse
QColor black(Qt::black);
QColor red(Qt::red);
QPen pen(black);
QBrush brush(red);
// C'est sur le painter qu'on applique une ligne
painter->setPen(pen);
painter->setBrush(brush);
painter->drawLine(10,10,150,150);
painter->end();
// Une fois la painter fermé on sauve le pixmap dans un fichier
pixmap->save("fichier.png");
```

Production de lettres en images

- Écrire un programme qui produit autant de fichiers que de

caractères ASCII visibles

- Chaque fichier sera une image au format png d'une taille paramétrable
- Chaque image représentera un caractère, d'une couleur paramétrable

dans un rectangle arrondi, lui même d'une couleur paramétrable,
entouré d'un trait d'épaisseur paramétrable et de couleur paramétrables

GraphicsScene

Présentation

- On utilise une scène pour y placer des objets graphiques
- La séquence théorique d'utilisation est :

```
QGraphicsScene scene;
scene.addText("Hello, world!");
QGraphicsView view(&scene);
view.show();
```

- La scène permet d'ajouter des éléments QGraphicsItem
- Ces items comprennent les objets suivants`QGraphicsEllipseItem`, `QGraphicsLineItem`,`QGraphicsPixmapItem`,`QGraphicsPolygonItem`, `QGraphicsRectItem` `QGraphicsSimpleTextItem` et`QGraphicsTextItem`
- L'intérêt d'une QGraphicsScene est sa capacité à gérer des millions d'Item et d'en connaître l'emplacement en quelques millisecondes

QGraphicsProxyWidget

- On peut également ajouter sur une scène des Widgets classiques via la classe `QGraphicsProxyWidget`

```
QPushButton *bouton = new QPushButton("Un bouton  !");
QGraphicsScene scene;
 QGraphicsProxyWidget *proxy = scene.addWidget(bouton);
proxy->setWidget(bouton);
QGraphicsView view(&scene);
view.show();
```

Utilisation à partir d'un QGraphicsView

- Dans le Designer, on dispose d'un objet QGraphicsView
- On passe du QGraphicsView au QGraphicsScene comme ceci :

```
m_graphicsscene=new QGraphicsScene(this);
 ui->graphicsView->setScene(m_graphicsscene);
m_graphicsscene->addText(QString("Exemple"));
```

- La scène permet d'envoyer des directives comme clear, addLine, addRect, addPolynom etc..
- AddRect(QRect,QPen,QBrush) les trois arguments sont :
  - un rectangle donné sous forme QRect ou x,y,largeur,longeur
  - un stylo d'une couleur donnée pour dessiner le contour
  - un motif pour remplir le contour du rectangle

```
QColor color(Qt::red)
QPen pen(color);
QBrush brushcolor(color,Qt::SolidPattern)
```

Coordonnées sur une scène

- l'axe X est l'axe horizontal orienté vers la droite et l'axe Y est l'axe vertical orienté vers le bas
- Afin de simplifier les conversions d'un système de coordonnées à un autre, Qt fournit des méthodes mapToScene() et mapFromScene() dans les classes QGraphicsView et QGraphicsItem qui permettent de retrouver les coordonnées d'un point dans la scène à partir des coordonnées dans la vue ou l'item et inversement.
- La scène s'arrange pour positionner les objets au milieu de la vue aussi on peut imposer de voir la scène non pas centrée mais avec un centrage défini par rapport à ses coordonnées par :

```
scene.setSceneRect(-150, -150, 300, 300);
```

QGraphicsScene

- Préparer une scène comprenant une image et des cercles numérotés
- Les cercles numérotés seront implémentés par la méthode QGraphicsItem
- Gérer le déplacement des cercles numérotés à la souris

Divers

Rubriques

- Classes et cast
- Feuille de style, animation et timer
- Processus et Thread
- QObject

Classes et cast

Extension un mode full screen

- Ce mode sert pour l'affichage en plein écran puisque dans ce mode là on force un widget à ne pas avoir de décoration de fenêtre et à être fille du desktop
- on procèdera ainsi

```
diaporamawidget = new DiaporamaWidget(0);
diaporamawidget->showFullScreen();
diaporamawidget->setFocusPolicy(Qt::StrongFocus);
```

- pour sortir du mode plein écran on appelle `showNormalScreen();`
- diaporamawidget n'étant pas fille de la fenêtre qui l'appelle (puisqu'elle est fille de la fenêtre desktop), aucun destructeur ne sera appelé en cas de destruction de la fenêtre parente. Attention aux pertes de mémoire.

Les casts en C++

- Le C++ offre les types de casts suivant static_cast
- static_cast permet une conversion de types compatibles, mais également de types double* vers long* ou de void* vers int* . Par contre un static_cast ne peut convertir un double* vers un float*.

```
void *vp;
long* lp = static_cast<long*>(vp);
```

- reinterpret_cast permet de faire des conversions de type double* vers float*

```
double* dp;
float* fp = reinterpret_cast<float*>(dp);
```

- const_cast permet de pointer sur un const type &i en modifiant sa valeur

```
int i = 65;
const int& r_i = i;
int& r_i2 = const_cast<int&>(r_i);
```

- les dynamic_cast sont des casts qui sont déterminés à l'exécution d'un programme comme le font les méthodes virtuelles

Les Cast en Qt

- le cast qobject_cast :

```
T qobject_cast(QObject *object)
```

renvoie un pointeur vers le type T ou vers une sous classe, si l'objet casté est de type T sinon il renvoie 0

```
QObject *obj = new QTimer;          // QTimer inherits QObject

QTimer *timer = qobject_cast<QTimer *>(obj);
  // timer == (QObject *)obj

  QAbstractButton *button = qobject_cast<QAbstractButton *>(obj);
  // button == 0
```

- Il est similaire au dynamic_cast mais ne requiert pas le support RTTI

Les pointeurs intelligents en Qt

- **QPointer** : pointeur léger indique si l'objet pointé a été détruit. Mais méfiance en multithread il peut renvoyer une valeur nulle, mais à l'instruction suivant le test, l'objet peut se voir détruit
- **QSharedDataPointer** : pointeur intelligent qui impose que la classe pointée possède les méthodes ref() et unref() qui augmente et décrémente le nombre de références sur l'objet. Si ref() renvoie null l'objet n'est plus pointé. Pour ce faire la classe pointée devra dériver de la classe QSharedData. Voir la classe QSharedPointerData
- **QExplicitlySharedDataPointer** :

Feuille de style, animation et timer

Feuille de style

- Une feuille de style CSS peut s'apliquer à toute une application via la méthode setStyleSheet(QString feuilledestyle) dans le fichier main.cpp

```
 QApplication a(argc, argv);
QFile file ("feuilledestyle.css");
file.open(QIODevice::ReadOnly|QIODevice::Text);
QString style=file.readAll();
 a.setStyleSheet(style);
```

- On peut également appliquer la méthode setStyleSheet au pointeur qApp() n'importe où dans l'application

```
qApp()->setStyleSheet(style);
```

- Une feuille de style peut s'appliquer également à un seul QWidget

Exemples de feuilles de style

- Application à un QLineWidget (chapitre documentation The Style Sheet Syntax)

```
QLineEdit { background: yellow }
```

- A plusieurs widgets

```
QPushButton, QLineEdit, QComboBox { color: red }
```

- On peut appliquer
  - à un nom d'objet par la syntaxeQPushButton#okButton
  - à un descendant : QDialog > QPushButton
- à un objet sans ses sous classes .QPushButton
- Certains widget disposent de sous controles : QComboBox::drop-down { image: url(dropdown.png) } ou d'états QPushButton:hover { color: white }

QAbstractAnimation

- La classe QAbstractAnimation sert de base à toute animation
- Deux classes dérivées permettent de gérer les animations
  - QVariantAnimation permettant de gérer les animations par QPropertyAnimation
  - QAnimationGroup permettant de gérer des animations parallèles ou séquentielles via les classes QParallelAnimationGroup et QSenquentialAnimationGroup
- La classe QPropertyAnimation prend deux arguments le widget et la propriétés que l'on veut animer
- Exemple d'animation d'un bouton :

```
QPushButton *button = new QPushButton("Animated Button",this);
 button->show();
 QPropertyAnimation *animation = new QPropertyAnimation  (button,"geometry");

animation->setDuration(10000);
animation->setStartValue(QRect(0, 0, 100, 30));
animation->setEndValue(QRect(250, 250, 100, 30));
animation->start();
```

Animation d'un objet

- La méthode QPropertyAnimation animation(&button, "geometry") permettait de jouer sur la propriété geometry ce qui est possible parce que la classe QWidget dispose d'une propriété geometry indiquant sa taille et sa position par rapport à l'objet parent.
- Si on veut créer une animation via une nouvelle propriété, il faut alors définir cette propriété

```
class MyGraphicsRectItem : public QObject, public QGraphicsRectItem
{
    Q_OBJECT
    Q_PROPERTY(QRectF geometry READ geometry WRITE setGeometry)
};
```

Groupement d'animation

- Les classes QParallelAnimation et QSequentialAnimation permettent de grouper des animations

```
QParallelAnimationGroup *group = new QParallelAnimationGroup;
group->addAnimation(anim1);
group->addAnimation(anim2);
group->start();
```

- L'animation parallèle jouera les deux animation ensemble, la séquentielle attendra que la première soit jouée avant de jouer la seconde.

QTimer

- La classe QTimer permet de déclencher un timer répétitif ou bien en une seule passe
- Timer Repetitif

```
QTimer *timer = new QTimer(this);
connect(timer, SIGNAL(timeout()), this, SLOT(update()));
timer->start(1000);
```

- Timer une seule fois

```
QTimer::singleShot(200, this, SLOT(updateCaption()));
```

Processus et Thread

QProcess

- Start (program, argument) permet de lancer un programme externe
- Une fois lancé il entre dans l'état Running et émet le signal started
- On peut écrire sur l'entrée standard du process par write() ou lire la sortie standard par read()
- A la fin du process le signal finished est émis.
- QProcess Signaux disponibles
  - void error (QProcess::ProcessError error)
  - void finished (int exitCode, QProcess::ExitStatus exitStatus)
  - void readyReadStandardError ()
  - void readyReadStandardOutput ()
  - void started ()
  - void stateChanged (QProcess::ProcessState newState)
  - ainsi que les signaux hérités de QIODevice

QProcess

- Charger les sources de la librairie Qt sur le site de Qt-Project au format tgz
- Ecrire un programme qui lance la décompression du programme par QProcess
- Au moment du démarrage de la décompression afficher le curseur d'attente de la souris
- A la fin de la décompression afficher une fenêtre comprenant le code

retour et remettre le curseur de la souris en mode normal

- Sous Windows : on téléchargera un utilitaire de décompression en ligne de commandes

QThread

- Si QProcess est très utile pour piloter un processus externe de façon efficace, les QThread permettent de piloter des processus écrits en Qt de façon plus fine :
  - en utilisant des sémaphores qui permettront de gérer la parallélisation par des mécanismes d'attente
  - en fournissant des mutext protégeant l'espace données et permettant la communication inter processus
- D'autre part :
  - Dans une application multithread, l'interface utilisateur graphique s'exécute dans son propre thread et le traitement a lieu dans un ou plusieurs threads distincts. De cette façon, l'interface utilisateur graphique des applications reste réactive, même pendant un traitement intensif.
  - Un autre avantage de l'environnement multitread réside dans le fait que les systèmes multiprocesseurs peuvent exécuter plusieurs threads simultanément sur différents processeurs, offrant ainsi de meilleures performances.

Mécanisme QThread

- Deux méthodes protégées qui obligent à définir une classe dérivée de QThread
  - `exec()` : entre dans la boucle d'évènement et attend un exit()
  - `run()` : est appelé après que `start()` a été appelé.
- On peut quitter un Thread par
  - `exit(code retour)`
  - `quit()` qui est équivalent à exit(0)
  - On évite de terminer par terminated qui est une terminaison violente
- Deux signaux sont appelés :
  - `finished()` : quand le thread se finit
  - `started()` : quand le thread démarre

Lancement d'un Thread

- Lancer un Thread avec un run qui boucle avec dans sa boucle une attente de 1 seconde et l'émission d'un signal
- Écrire une interface qui permet de lancer le Thread, de l'arrêter, de suivre sa progression par un progress bar et de quitter le programme.
- Pour le moment on n'utilise qu'un thread

Méthode QObjet relatives aux threads

- La méthode suivante indique que l'objet et ses enfants feront partie du Thread

```
myObject->moveToThread(QApplication::instance()->thread());
```

- Pour cela l'objet ne doit pas avoir de parent
- Cette fonction a pour effet de pousser un objet d'un thread vers un autre
- La méthode QThread *QObject::thread() retourne le thread dans lequel l'objet vit

QObject

La classe QMetaMethod

- L'énum QMetaMethod::Private, Protected, Public: indique la portée
- L'énumération QMetaMethod::MethodType: Method, Signal, Slot, Constructor
- int methodIndex() : index de la méthode
- MethodType::methodType()
- QByteArray::methodName()
- int parameterCount()
- QList parameterNames()
- QList parameterTypes
- Pour chaque méthode, on dispose de ces informations contenues dans la classe QObject

La classe metaObject

- La classe metaObject contient des méta informations à propos des Objets Qt
- className() : nom de la classe
- superClass() : classe mère
- userProperty() : propriété qui ont le flag user à true. Par exemple la méthode text de QLineEdit a son flag user à true parce qu'un utilisateur peut le modifier
- isWritable() : la propriété est modifiable
- isValid() : la propriété existe
- name() : nom de la propriété
- ...
- Ces méta informations sont utilisées par le Designer, pour connaître les noms des slot appelant, l'émetteur d'un signal etc..

Les properties

- Pour un Objet les propriétés sont déclarées par la macro

```
Q_PROPERTY(type name
           (READ getFunction [WRITE setFunction] |
            MEMBER memberName [(READ getFunction | WRITE setFunction)])
           [RESET resetFunction]
           [NOTIFY notifySignal]
           [REVISION int]
           [DESIGNABLE bool]
           [SCRIPTABLE bool]
           [STORED bool]
           [USER bool]
           [CONSTANT]
           [FINAL])
```

- On retrouve les fonctions des méta Objets (iswritagble : WRITE)
- On retrouve ces propriétés dans les déclarations des classes par exemple

```
Q_PROPERTY(QSize iconSize READ iconSize WRITE setIconSize)
```

- Ces propriétés servent à renseigner les MetaObjets.

La classe QObject

- dispose des connects (de signal à slot ou de signal à signal) et disconnect (sur un signal ou sur un objet)
- QObject::findChild(const QString &name = QString(), Qt::FindChildOptions options = Qt::FindChildrenRecursively) const ;

```
QPushButton *button = parentWidget->findChild<QPushButton *>("button1");
QListWidget *list = parentWidget->findChild<QListWidget *>(QString(), Qt::Find
```

- QList QObject::findChildren(const QString &name = QString(), Qt::FindChildOptions options = Qt::FindChildrenRecursively) const

```
QList<QWidget *> widgets = parentWidget.findChildren<QWidget *>("widgetname");
```

- const QMetaObject *QObject::metaObject()

```
QObject *obj = new QPushButton;
obj->metaObject()->className();          // returns "QPushButton"
```

- obj->metaObject()->className(); // returns “QPushButton”

QObject et signal

- int QObject::receivers(const char *signal) const : nombre de receveurs d'un signal
- bool QObject::isSignalConnected(const QMetaMethod &signal) const : le signal est connecté à au moins un receveur*
- QObject *QObject::sender() const : pointeur vers l'objet qui a envoyé le signal

Méthode QObjet relatives aux threads

- La méthode suivante indique que l'objet et ses enfants feront partie du Thread

```
myObject->moveToThread(QApplication::instance()->thread());
```

- Pour cela l'objet ne doit pas avoir de parent
- Cette fonction a pour effet de pousser un objet d'un thread vers un autre
- La méthode QThread *QObject::thread() retourne le thread dans lequel l'objet vit