

Introduction Qt

HAI9111 – cours I

Qu'est-ce que Qt ?

“Qt is a cross platform development framework written in C++.”

- C++ framework – liens vers les autres languages
- Python, Ruby, C#, etcetera
- Originellement pour les interfaces utilisateurs – maintenant pour tout
- Bases de données, XML, WebKit, multimédia, networking, OpenGL, scripting, non-GUI...

Qu'est-ce que Qt ?

- A base de **modules**
- Style d'API commun

Modules

GUI

Essentials

Widgets C++ Native LAF Layouts Styles OpenGL	Qt Quick QML Controls Layouts Styles OpenGL	WebEngine + WebView HTML 5 Hybrid UIs
Core Processes Threads IPC Containers I/O Strings Etc.	Multimedia Audio Video Radio Camera	Network HTTP FTP TCP/UDP SSL
	Sql SQL and Oracle databases	Qt Test

non-GUI

Add-ons

Charts	
SVG	Data Visualization
Canvas 3D	Virtual Keyboard
Serial Port	Bluetooth
Positioning	Concurrency
Printing	Scripting
NFC	Platform Extras
XML	Sensors
Image formats	In-App Purchasing

Qu'est-ce que Qt ?

- Qt étend le C++ avec des macros et l'introspection

```
foreach (int value, intList) { }  
  
QObject *o = new QPushButton;  
o->metaObject()->className(); // returns QPushButton  
connect(button, SIGNAL(clicked()), window, SLOT(close()));
```

- Tout le code est du C++

L'intérêt de Qt

- Un code → Applications multi-plateformes
 - Windows, Linux, Mac+Mobile (Android/iOS)
- Applications natives
- Facile de (ré)utiliser des API, développement efficace, open-source



Hello World

```
#include <QApplication>
#include <QLabel>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QLabel *label = new QLabel();
    label->setObjectName(QString::fromUtf8("label"));
    label->setGeometry(QRect(100, 100, 200, 100));
    label->setText("Hello World");
    label->show();
    return app.exec();
}
```

qmake

- Compilation : Makefile + outils standard
- Projet et compilation : décrits dans un fichier « .pro »
- qmake → génère automatiquement un Makefile
- IDE : QtCreator
- Générer un « .pro »
 - qmake en mode génération de projet
 - QtCreator
 - Peut-être complexe (multi-plateforme=
 - [qmake user guide](#)

qmake

```
Qt1$ qmake -project
```

- Génère un projet qt simple

```
#####  
# Automatically generated by qmake (3.0) Mon Nov 14 08:32:53 2016  
#####  
  
TEMPLATE = app  
TARGET = Qt1  
INCLUDEPATH += .  
# note we need to add widgets module for gui  
QT += widgets  
# Input  
SOURCES += main.cpp
```

- Ensuite taper « make » pour compiler

Variables qmake utiles

- **TEMPLATE** : définit le type de projet
 - app, vcapp lib, vclib subdirs
- **TARGET** : nom de l'exécutable (défaut = nom du .pro)
- **QT** : module Qt-spécifiques et leurs dépendances définies dans mkspecs/modules
 - QT += webkit sql network charts

Variables qmake utiles

- CONFIG

- Configuration de projet ou option du compilateur
- Utilisée en interne par qmake
- Peut référer à un fichier de config .prf dans mkspecs/features
- Des configurations personnalisées : `CONFIG += myFeatures`

- Astuce :

- variables additionnelles et valeurs en ligne de commande :
`qmake "CONFIG += debug"`

Variables qmake utiles

- INCLUDEPATH and DEPENDPATH
 - Chemin pour les inclusions (option -I)
- RESOURCES
 - Collection de ressources fichiers (.qrc) à inclure dans le build
 - E.g. : images de texture, shaders...
- LIBS
 - option -L du compilateur
- DEFINES
 - option -D du compilateur (#if FLAG)
- QMAKE_CXX compilateur à utiliser
 - QMAKE_CXX=/usr/bin/g++
- \$\$system([command]) commande externe

```
QMAKE_CXXFLAGS+=$$system(sdl2-config --cflags)
```

```
LIBS+=$$system(sdl2-config --libs)
```

CMake

- CMake complètement intégré à Qt
- Spécification de l'emplacement de Qt install nécessaire
 - E.g: variable d'environnement CMAKE_MODULE_PATH vers C:\Qt\5.12.6\

CMakeLists

```
cmake_minimum_required(VERSION 3.12)
# Name of the project
project(Qt1Build)
# This is the name of the Exe change this and it will change
everywhere
set(TargetName Qt1)
# Instruct CMake to run moc automatically when needed (Qt
projects only)
set(CMAKE_AUTOMOC ON)
# Run the uic tools and search in the ui folder set(CMAKE_AUTOUIC
ON)
set(CMAKE_AUTOUIC_SEARCH_PATHS ${PROJECT_SOURCE_DIR}/ui)
# find Qt libs
find_package(Qt5 COMPONENTS Widgets REQUIRED)
# use C++ 17
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS ON)
```

Modules Qt et inclusion

- Modules :
 - QtCore, QtGui, QtWidgets, QtXml, QtSgl, QtNetwork...
- Activer un module dans le qmake.pro :

```
QT += network
```

- Défaut : QtCore et QtGui
- Toutes les classes Qt ont un fichier header

```
#include <QLabel>  
#include <QtWidgets/QLabel>
```

- Tous les modules ont un fichier header

```
#include <QtGui>
```

Inclusion et temps de compilation

- Modules :

```
#include <QtGui>
```

- Header précompilé et le compilateur :

- Non-supporté : compilation plus longue
- Supporté : accélération (Windows, MacOSX, Unix)

- Class :

```
#include <QtLabel>
```

- Réduit le temps de compilation

- Forward declaration (`class QLabel;`)
- Placer les modules includes avant les autres


```

graph TD
    QObject[QObject] --> QObjectStaticMethods[QObject::Static Methods]
    QObject --> QObjectPrivateMethods[QObject::Private Methods]
    QObject --> QObjectProtectedMethods[QObject::Protected Methods]
    QObject --> QObjectPublicMethods[QObject::Public Methods]
    QObject --> QObjectPrivateStaticMethods[QObject::Private Static Methods]
    QObject --> QObjectProtectedStaticMethods[QObject::Protected Static Methods]
    QObject --> QObjectPublicStaticMethods[QObject::Public Static Methods]
    QObject --> QObjectPrivateProtectedMethods[QObject::Private Protected Methods]
    QObject --> QObjectProtectedProtectedMethods[QObject::Protected Protected Methods]
    QObject --> QObjectPublicProtectedMethods[QObject::Public Protected Methods]
    QObject --> QObjectPrivatePublicMethods[QObject::Private Public Methods]
    QObject --> QObjectProtectedPublicMethods[QObject::Protected Public Methods]
    QObject --> QObjectPublicPublicMethods[QObject::Public Public Methods]
    QObject --> QObjectPrivatePrivateMethods[QObject::Private Private Methods]
    QObject --> QObjectProtectedPrivateMethods[QObject::Protected Private Methods]
    QObject --> QObjectPublicPrivateMethods[QObject::Public Private Methods]
    QObject --> QObjectPrivateProtectedPrivateMethods[QObject::Private Protected Private Methods]
    QObject --> QObjectProtectedProtectedPrivateMethods[QObject::Protected Protected Private Methods]
    QObject --> QObjectPublicProtectedPrivateMethods[QObject::Public Protected Private Methods]
    QObject --> QObjectPrivatePublicPrivateMethods[QObject::Private Public Private Methods]
    QObject --> QObjectProtectedPublicPrivateMethods[QObject::Protected Public Private Methods]
    QObject --> QObjectPublicPublicPrivateMethods[QObject::Public Public Private Methods]
    QObject --> QObjectPrivatePrivatePrivateMethods[QObject::Private Private Private Methods]
    QObject --> QObjectProtectedPrivatePrivateMethods[QObject::Protected Private Private Methods]
    QObject --> QObjectPublicPrivatePrivateMethods[QObject::Public Private Private Methods]
  
```

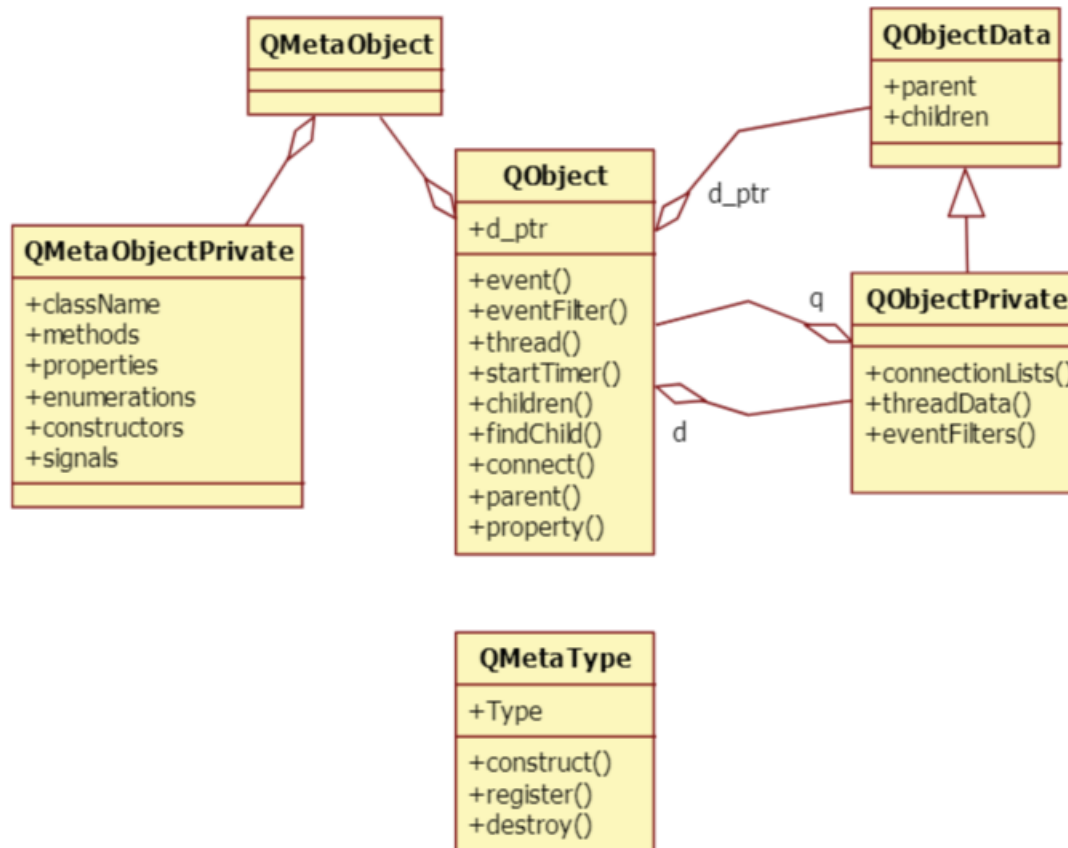
- Class de base de presque toutes les class Qt et Widgets
- Contient presque tous les composants faisant la particularité de Qt:
 - events, signals, slots
 - Propriétés
 - Gestion mémoire etc...

QObject

- Base de la plupart des Qt classes.
- Exemple d'exceptions :
 - Classes devant être « légères » (primitives graphiques)
 - Data containers (QString, QList, QChar, etc)
 - Devant être copié (! QObject ne le peuvent pas)

QObject : propriétés

- Nom : `QObject::objectName`
- Placés dans une hiérarchie d'instances de `QObject`
- Connectés à d'autres instances



(type) introspection

- Définition : en informatique, capacité à déterminer le type de l'objet à l'exécution
- En C++ : mots clés typeid et dynamic_cast

- typeid : retrouve un objet [std::type_info](#) décrivant le type

```
std::cout << typeid(obj).name() << '\n';
```

- dynamic_cast : permet de tester si un objet est d'un type

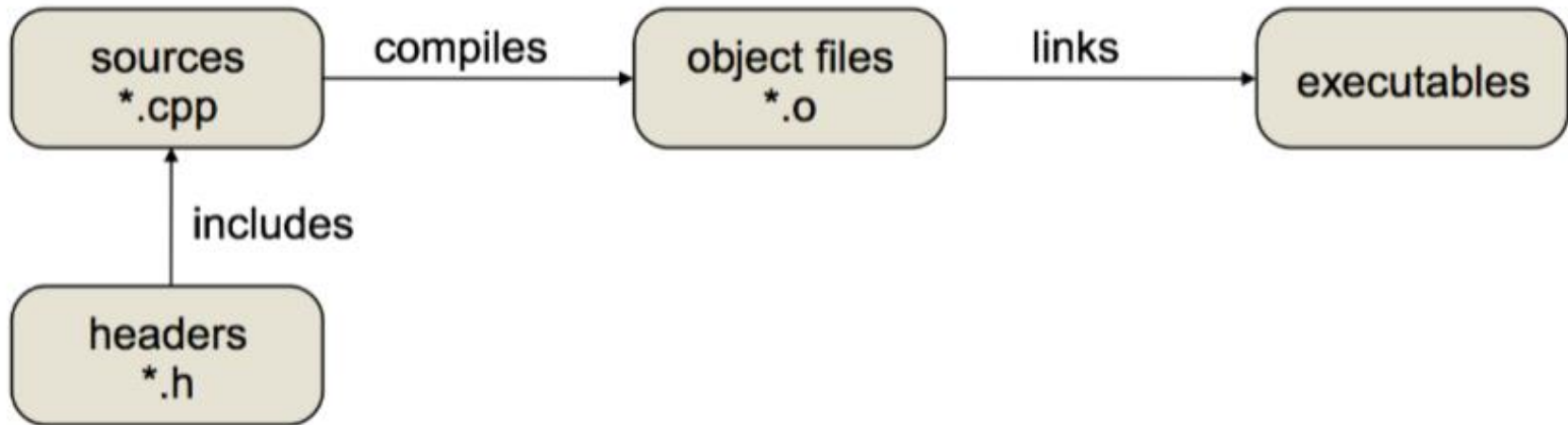
```
Person* p = dynamic_cast<Person *>(obj);  
if (p != nullptr) {  
    p->walk();  
}
```

Meta données

- Qt implemente l'introspection en C++
- Tous les QObject ont un meta objet
- Le meta object connaît
 - class name ([QObject::className](#))
 - inheritance ([QObject::inherits](#))
 - properties
 - signals and slots
 - general information ([QObject::classInfo](#))

Méta données

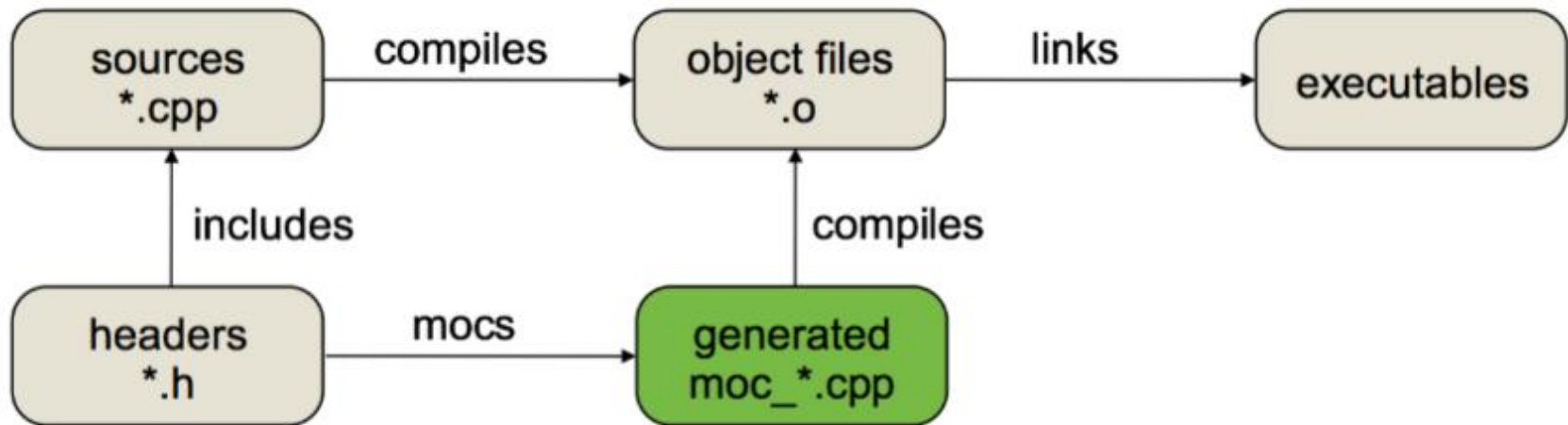
Ordinary C++ Build Process



- The meta data is gathered at compile time by the meta object compiler, [moc](#).

Méta données

- Les méta-données sont collectées à la compilation par le compilateur de meta-objets, **moc**



- Le moc collecte les données des headers.

Méta-données

- Que cherche le moc ?

```
class MyClass : public QObject {
    Q_OBJECT
    Q_CLASSINFO("author", "John Macey")
public:
    MyClass(const Foo &foo, QObject *parent=0);
    Foo foo() const;
public slots:
    void setFoo( const Foo &foo );
signals:
    void fooChanged( Foo );
private:
    Foo m_foo;
};
```


Introspection

- Résultat : les classes se « connaissent elles-mêmes » à l'exécution
- Connaître les types présents
- Identifier des classes spécifiques
- Passer des messages entre les classes ou multi-caster vers plusieurs classes
- Le cœur de la communication inter objets en Qt : système de signals et slots

Propriétés

- QObject : propriétés avec des « getter » et « setter »

```
class QLabel : public QFrame
{
    Q_OBJECT
    Q_PROPERTY(QString text READ text WRITE setText)
public:
    QString text() const;
public slots:
    void setText(const QString &);
};
```

- Convention de nom :
 - Color, setColor
 - Booléens : isEnabled, setEnabled

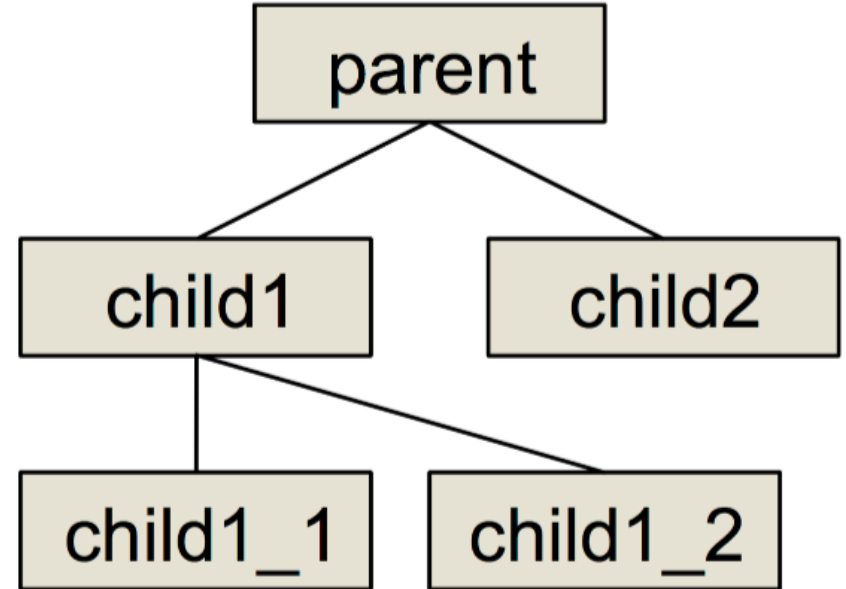
Propriétés

- Possible d'y accéder de différentes façons
- Possible d'ajouter des propriétés à l'exécution
- Permet le « object tagging »
 - E.g. READ, WRITE, READ/WRITE

Gestion de la mémoire

- Les QObject ont un parent et des enfants
- Suppression d'un parent, il supprime ses enfants

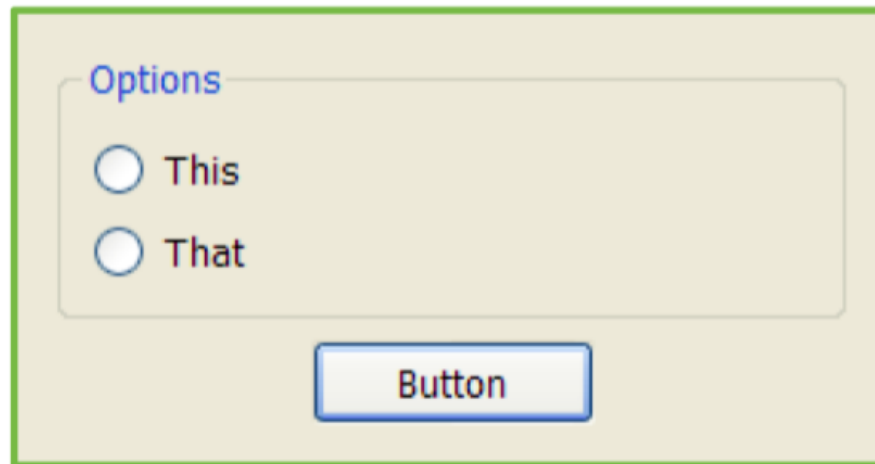
```
QObject *parent = new QObject();  
QObject *child1 = new QObject(parent);  
QObject *child2 = new QObject(parent);  
QObject *child1_1 = new QObject(child1);  
QObject *child1_2 = new QObject(child1);  
delete parent;
```



Gestion de la mémoire

- Implementation de hiérarchie visuelle

```
QDialog *parent = new QDialog();  
QGroupBox *box = new QGroupBox(parent);  
QPushButton *button = new QPushButton(parent);  
QRadioButton *option1 = new QRadioButton(box);  
QRadioButton *option2 = new QRadioButton(box);  
delete parent;
```

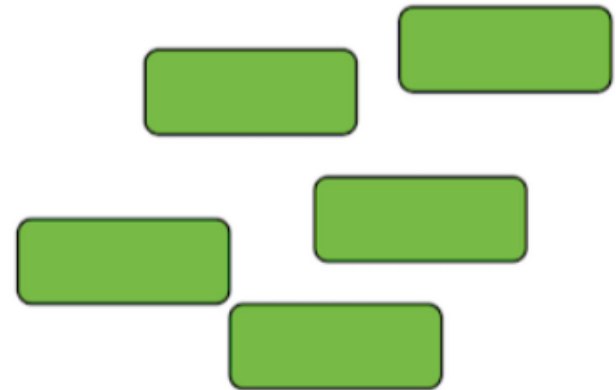


Tas (heap)

- En utilisant `new` et `delete`, la mémoire est allouée sur le tas.
- La mémoire du tas doit être explicitement libérée en utilisant `delete` pour éviter les fuites mémoires.
- Les objets peuvent vivre aussi longtemps que nécessaire

`new` 

Construction



Destruction

`delete` 

Pile d'exécution (stack)

- Les variables locales sont allouées sur la pile
- Sortie de « scope » : variables de la pile détruites

int a



Construction

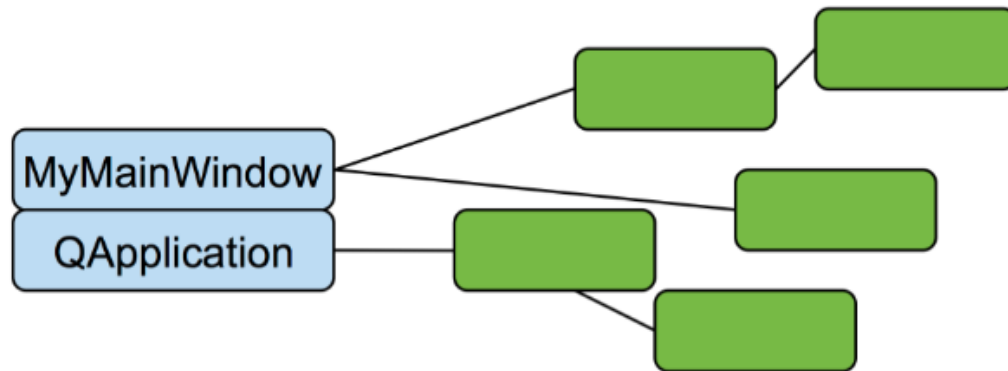


Destruction



Tas et pile

- Pour avoir une gestion automatique de mémoire, seulement le parent doit être alloué sur la pile



```
int main(int argc, char **argv)
{
    QApplication a(argc, argv);
    MyMainWindow w; // stack
    w.show();
    return a.exec();
}
```

```
MyMainWindow::MyMainWindow(...)
{
    new QLabel(this); // heap
    new ...
}
```


Constructeurs

- parent = 0 : valeur par défaut pour presque tous les QObject

```
QObject(QObject *parent=0);
```

- Premier argument avec une valeur par défaut
- Parents des QWidget sont des QWidget
- Nombreux constructeurs

```
QPushButton(QWidget *parent=0);  
QPushButton(const QString &text, QWidget *parent=0);  
QPushButton(const QIcon &icon, const QString &text, QWidget *parent=0);
```

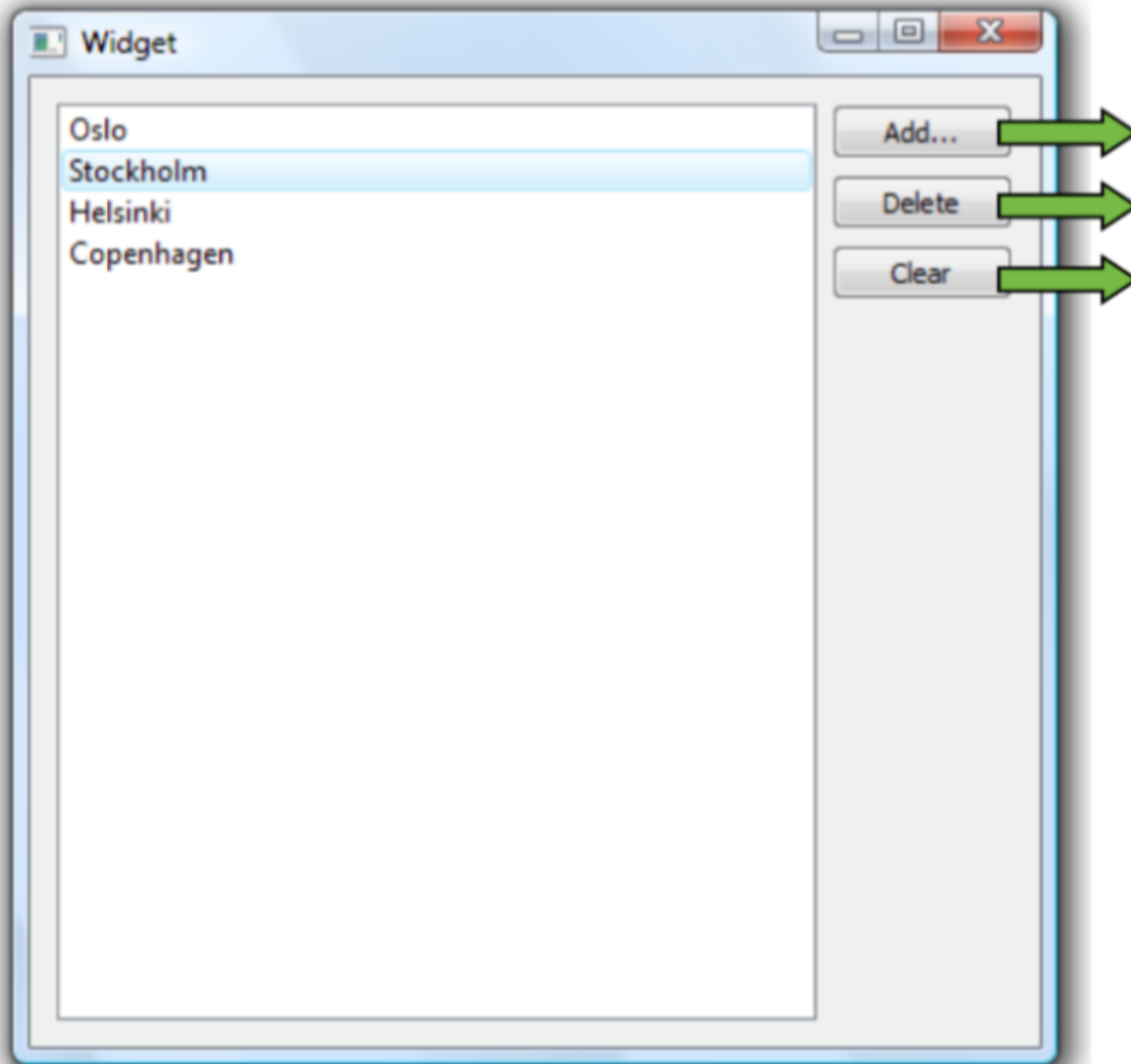
Constructeur

- Création de votre QObject
- Toujours autoriser le parent=0 (ou nullptr)
- Respecter la convention :
 - Constructeur avec seulement le parent
 - Avoir parent comme 1^{er} argument avec une valeur par défaut
 - Prévoir plusieurs constructeurs : évite de passer des variables à 0 (null) ou invalides (e.g. QString()) comme arguments

Signals et Slots

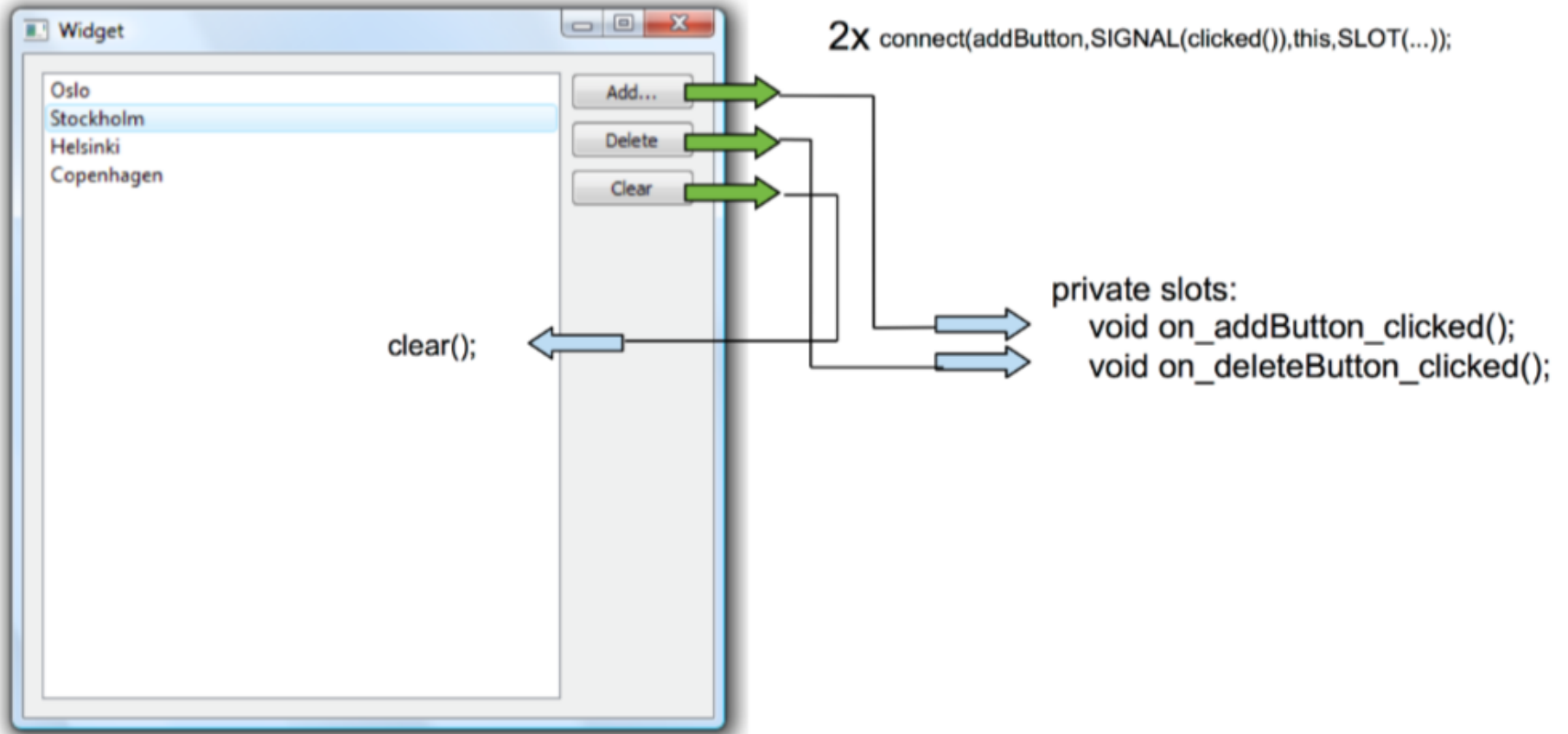
- Lient dynamiquement les évènements et les changements d'états avec des réactions
- Le cœur de Qt
- Basé sur le pattern de l'observateur

Signals et Slots en action



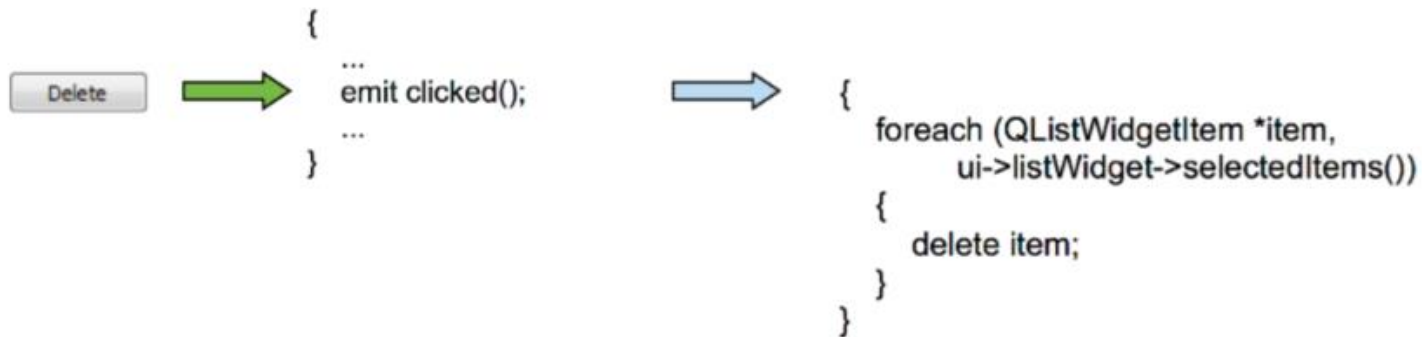
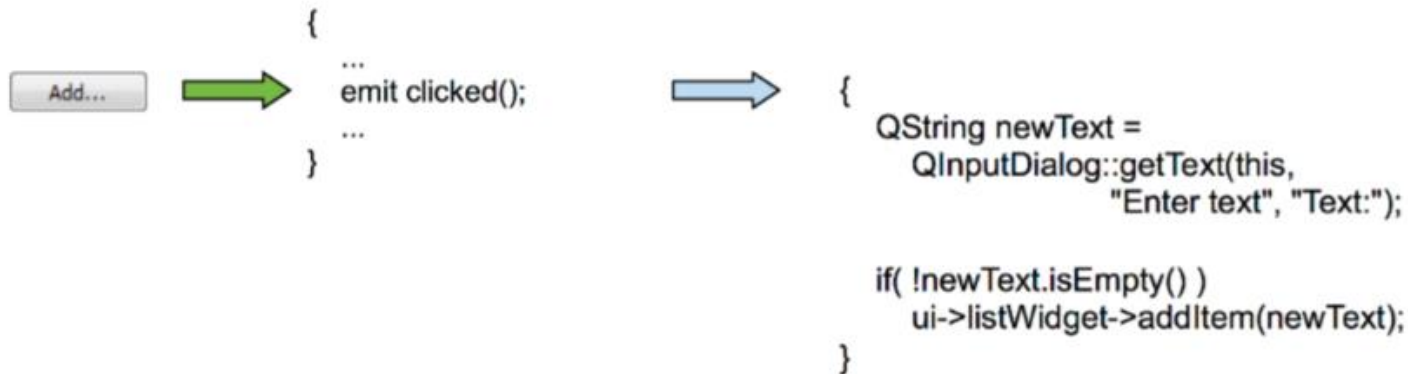
emit clicked();

Signals et Slots en action



```
connect(clearButton,SIGNAL(clicked()),listWidget,SLOT(clear()));
```

Signals et Slots en action



Oslo
Stockholm
Helsinki
Copenhagen

Signals et Slots vs Callbacks

- Callback : pointeur vers une fonction appelée quand un évènement se produit, toutes fonction peuvent être attribuées
 - Pas de « type-safety »
 - Fonctionne toujours en appel direct
- Signals et Slots sont plus dynamiques
 - Mécanisme plus générique
 - Plus facile de connecter 2 classes
 - Moins de connaissances partagées entres les classes impliquées

Qu'est-ce qu'un slot ?

- Définit dans une section slot

```
public slots:  
    void aPublicSlot();  
protected slots:  
    void aProtectedSlot();  
private slots:  
    void aPrivateSlot();
```

- Peut retourner une valeur mais pas par connexion
- Nombre arbitraire de signaux peuvent être connectés

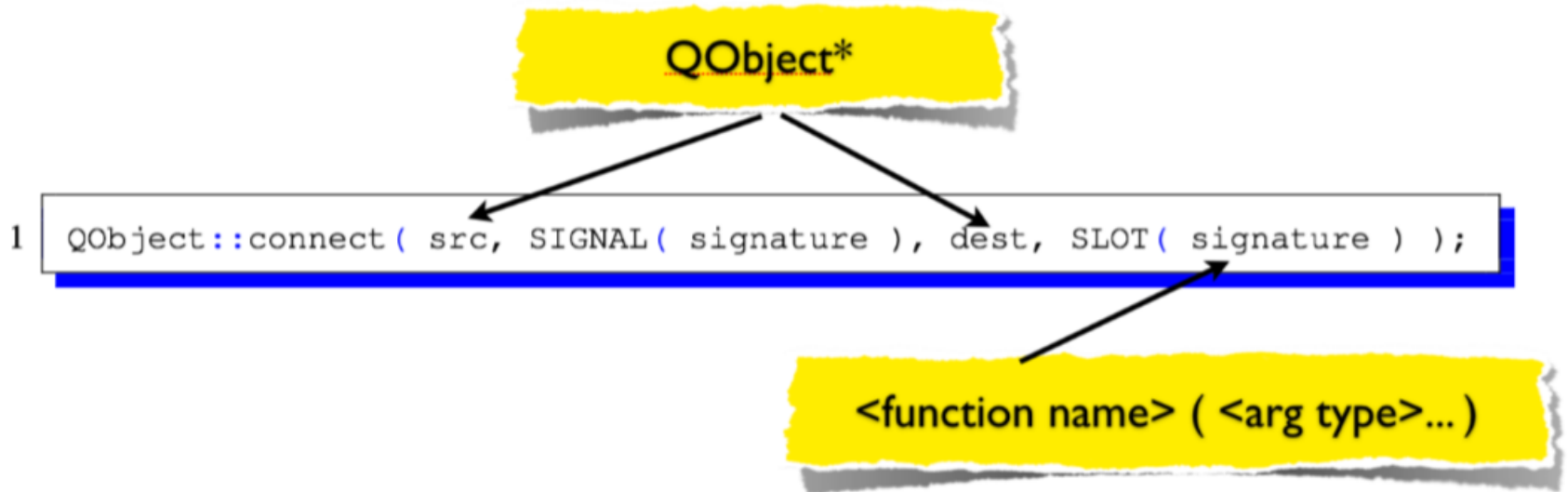
```
connect(src, SIGNAL(sig()), dest, SLOT(slot()));
```

- Implémentation comme une méthode ordinaire
- Peut être appelé comme une méthode ordinaire

Qu'est-ce qu'un signal

- Définit dans la section signals
 - Retourne toujours void
 - Ne doit pas forcément être implémenté
 - Le moc fournit une implémentation
- Connexion possible à un nombre arbitraire de slots
 - Résultant souvent en un appel direct
 - Peut être un évènement entre des threads ou sur un socket
 - Les slots sont activés dans un ordre arbitraire
- Activation : mot clé emit




Connexion



- Signature : nom de la fonction et type des arguments
- Pas de noms de variables ni de valeurs
- Type « maison » réduit la réutilisation

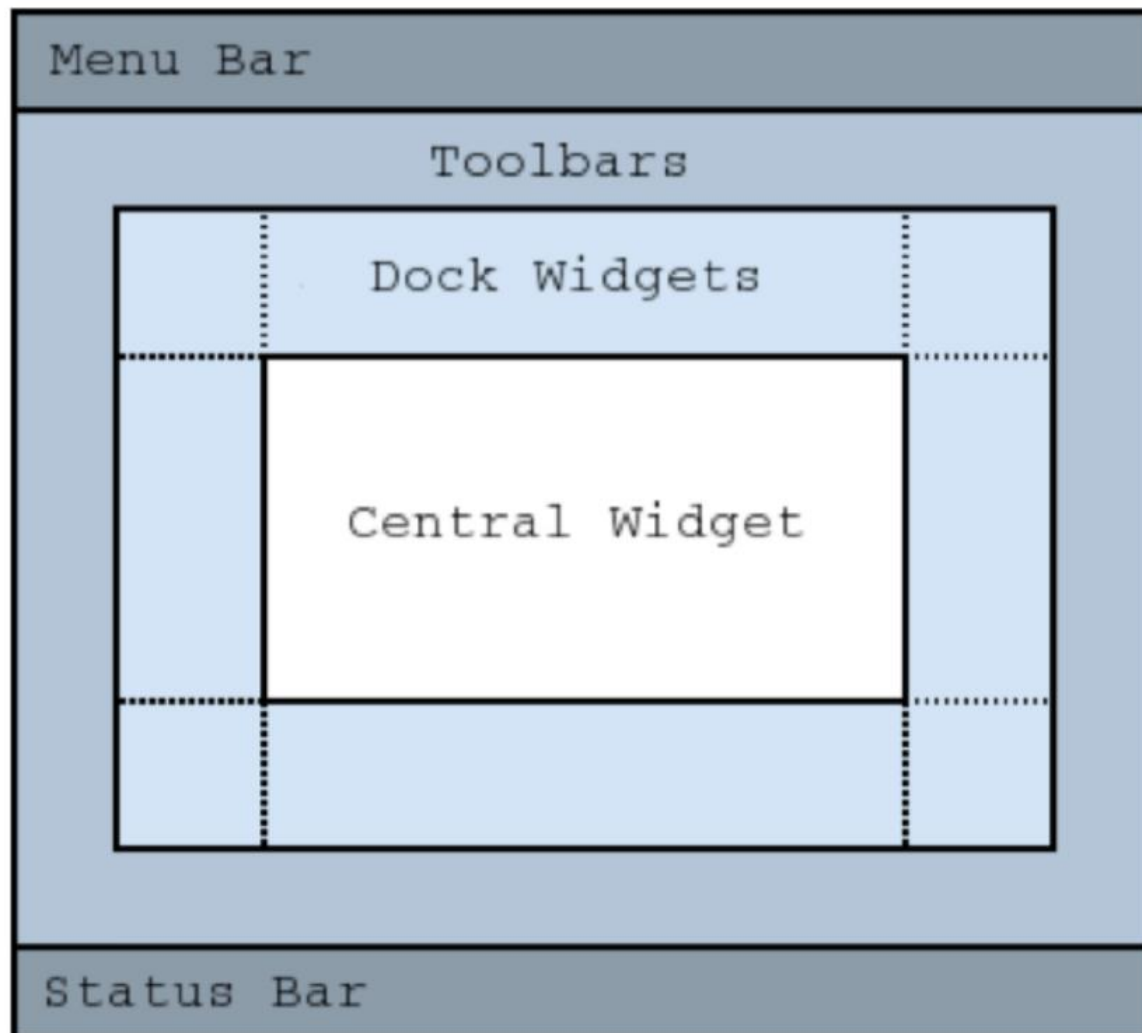
Connexion

- Qt peut ignorer des arguments pas créer des valeurs

Signals		Slots
rangeChanged(int,int)	—————	setRange(int,int)
rangeChanged(int,int)	—————	setValue(int)
rangeChanged(int,int)	—————	updateDialog()
valueChanged(int)	———  ———	setRange(int,int)
valueChanged(int)	—————	setValue(int)
valueChanged(int)	—————	updateDialog()
textChanged(QString)	———  ———	setValue(int)
clicked()	———  ———	setValue(int)
clicked()	—————	updateDialog()

La classe QMainWindow

- Fournit un framework pour construire une interface utilisateur
- QMainWindow et classes associées : gestion de fenêtre de Qt
- A son propre « layout », on peut ajouter QToolBars, QDockWidget, QMenu et QStatusBar
- Layout : aire centrale peut être n'importe quel type de widget



```
#include <QApplication>
#include <QMainWindow>
#include <QWidget>
int main(int argc, char *argv[])
{
    // create the main Qt app
    QApplication app(argc, argv);
    // create a main window widget
    QMainWindow *mainwin = new QMainWindow();
    // set the name
    mainwin->setObjectName(QString::fromUtf8("MainWindow"));
    // set the size
    mainwin->resize(200,200);
    // set the title of the window
    mainwin->setWindowTitle("A MainWindow App");
    // create a central widget with the main window as the parent
    QWidget *centralwidget = new QWidget(mainwin);
    // create a push button with the central widget as the parent
    QPushButton *button = new QPushButton(centralwidget);
    // set the name
    button->setObjectName(QString::fromUtf8("button"));
    // set the geometry
    button->setGeometry(QRect(10, 80, 100, 32));
    // set the text of the button
    button->setText("Button");
    // set the central widget for the main window
    mainwin->setCentralWidget(centralwidget);
    mainwin->show(); // show the window
    return app.exec(); // run the application
}
```

```
#include <QApplication>
#include <QMainWindow>
#include <QtWebEngineWidgets>
#include <QPushButton>
#include <QToolBar>
int main(int argc, char *argv[])
{
    // make an instance of the QApplication
    QApplication a(argc, argv);
    // Create a new MainWindow
    QMainWindow w;
    QToolBar *toolbar= new QToolBar();
    QPushButton *back= new QPushButton("back");
    QPushButton *fwd= new QPushButton("fwd");
    toolbar->addWidget(back);
    toolbar->addWidget(fwd);
    w.addToolBar(toolbar);
    w.addToolBar(toolbar);
    QWebEngineView *page = new QWebEngineView();
    page->load(QUrl("http://www.google.co.uk"));
    QObject::connect(back, SIGNAL(clicked()), page, SLOT(back()));
    QObject::connect(fwd, SIGNAL(clicked()), page, SLOT(forward()));
    w.setCentralWidget(page);
    w.resize(1024, 720);
    // show it w.show();
    // hand control over to Qt framework
    return a.exec();
}
```

Etendre la QMainWindow

- Exemple suivant : création de notre classe MainWindow
- Etend le parent QMainWindow et ajoute 2 slots pour gérer les entrées clavier et le changement de taille de fenêtre

MainWindow

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H
#include <QMainWindow>
#include <QKeyEvent>
/// @file MainWindow.h
class MainWindow : public QMainWindow
{
    Q_OBJECT
protected :
    /// @brief override the keyPressEvent inherited from QObject so we can
    handle key presses.
    /// @param [in] _event the event to process
    void keyPressEvent(QKeyEvent * _event);
    /// @brief override the resizeEvent inherited from QObject so we can handle
    key presses.
    /// @param [in] _event the event to process
    void resizeEvent(QResizeEvent * _event);
public:
    /// @brief constructor
    /// @param parent the parent window the for this window
    MainWindow(QWidget * _parent = 0);
    /// @brief dtor free up the GLWindow and all resources
    ~MainWindow();
private slots :

private:

};
#endif
// MAINWINDOW_H
```

Constructeur

```
MainWindow::MainWindow(QWidget *_parent ): QMainWindow(_parent)
{
    resize(QSize(1024,720));
    setWindowTitle(QString("Extending a MainWindow Class"));
}
```

Evènements clavier

```
void MainWindow::keyPressEvent(QKeyEvent *_event)
{
    // method called every time the main window receives a key event.
    // we then switch on the key value and set the camera in the GLWindow
    switch (_event->key()) {
        case Qt::Key_Escape : QApplication::exit(EXIT_SUCCESS);
        break;
        default : break;
    }
}
```

Changement de taille

- `resizeEvent` : appelé lorsque la taille de la fenêtre est changée
- Changer le titre indiquant la taille

```
void MainWindow::resizeEvent ( QResizeEvent * _event )
{
    QSize size=_event->size();
    QString title=QString("Extending a MainWindow Class size is %1 %2")
                    .arg( size.width())
                    .arg( size.height());
    this->setWindowTitle(title);
}
```

QString

- Essaie d'être la version modern de la classe string
- Stock des « Unicode strings » capables de représenter presque tous les systèmes d'écriture utilisés
- Permet les conversions entre les différents encodages
- API pratique pour l'inspection des chaînes de caractères et pour leur modification

QString

- Trois méthodes de construction
 - Opérateur +

```
QString res = "Hello " + name + ", the value is " + QString::number(42);
```

- QStringBuilder

```
QString res = "Hello " % name % ", the value is " % QString::number(42);
```

- Arg méthode

```
QString res = QString("Hello %1, the value is %2")  
                .arg(name)  
                .arg(42);
```

QStringBuiler

- Opérateur + demande de nombreuses allocations mémoire et vérifie la taille des chaînes
- Meilleure façon : QStringBuilder et l'opérateur %
- Le string builder collecte toutes les chaînes, évalue leur longueur avant de les joindre en une seule fois → une seule allocation mémoire