Université de la Rochelle
Licence d'Informatique - IHM (INFO132601)

TD 3 - *Programmation IHM haut niveau avec Qt*

© B. Besserer, R. Péteri

Année universitaire 2016-2017

1 Gestion de géométrie avec Qt

On souhaite écrire un programme Qt qui produit une fenêtre similaire à celle de gauche (type des widgets, position des widgets), fenêtre qui devra se comporter lors du redimensionnement (Resize-Event) afin de produire la fenêtre illustrée ci-dessous à droite.

On tiendra compte des espacements supérieurs et inférieurs entre le bord de la fenêtre et les Widgets. Le code produisant cette fenêtre sera directement écrit dans la routine `main()` ; il ne sera pas nécessaire d'implémenter une classe Qt dérivée. Il n'est pas nécessaire d'écrire un fichier de définition (.h) pour cet exercice.

```
#include <qapplication.h>
#include <qwidget.h>
#include <qlayout.h>
#include <qlabel.h>
#include <qcheckbox.h>
#include <qpushbutton.h>
void main(int argc, char ** argv) {
QApplication app(argc, argv);

// Création d'un widget générique de fond, qui sera parent de tous les autres widgets
QWidget widget;
// Utilisation d'une grille permettant d'ordonner tous les autres widgets
QGridLayout *l = new QGridLayout(...);
// Mise en place de la gestion de géométrie
// On autorise le redimensionnement pour certaines colonnes
l->setRowStretch(0, 1);
l->setColumnStretch(0, 1);
...
...
// a gauche, au milieu de la colonne, les 3 boites a cocher
QCheckBox *c1 = new QCheckBox("Pomme", &widget);
...
// a droite les 3 boutons poussoir
QPushButton *b1 = new QPushButton("Terminer", &widget);
```

```
...
// Générer et afficher le widget principal
app.setMainWidget(&widget);
widget.show();
return app.exec();
}
```

1. Quel est l'argument du constructeur de la classe `QGridLayout` et à quoi sert-il ?
2. Compléter le programme ci-dessus afin de réaliser les fonctionnalités souhaitées

```
#include <qapplication.h>
#include <qwidget.h>
#include <qlayout.h>
#include <qlabel.h>
#include <qcheckbox.h>
#include <qpushbutton.h>

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

// Création d'un widget générique de fond, qui sera parent de tous les autres widgets
QWidget widget;
// Utilisation d'une grille permettant d'ordonner tous les autres widgets
QGridLayout *l = new QGridLayout(&widget);
// Mise en place de la gestion de géométrie
// On autorise le redimensionnement pour certaines colonnes
l->setRowStretch(0, 1);
l->setRowStretch(4, 1);
l->setColumnStretch(1, 1);
// a gauche, au milieu de la colonne, les 3 boites a cocher
QCheckBox *c1 = new QCheckBox("Pomme", &widget);
l->addWidget(c1, 1, 0);
QCheckBox *c2 = new QCheckBox("Poire", &widget);
l->addWidget(c2, 2, 0);
QCheckBox *c3 = new QCheckBox("Banane", &widget);
l->addWidget(c3, 3, 0);
// a droite, au milieun les 3 boutons poussoir
QPushButton *b1 = new QPushButton("Terminer", &widget);
l->addWidget(b1, 1, 2);
QPushButton *b2 = new QPushButton("Annuler", &widget);
l->addWidget(b2, 2, 2);
QPushButton *b3 = new QPushButton("Detruire", &widget);
l->addWidget(b3, 3, 2);

// Afficher le widget principal
widget.show();
return app.exec();
}
```

3. Comment pourrait-on faire pour afficher une pop-up d'information (classe `QMessageBox`, méthode `setText`) chaque fois que la fenêtre principale est redimensionnée ?
   The event is QResizeEvent it will be send to QWidget : :resizeEvent() - a virtual function, which can be reimplemented in your widget.

```
#include <QtGui>

class myWindow:public QWidget
{
  public:
    myWindow():QWidget()
    {
              setWindowTitle("QWidget Resize Event");
                 };
    ~myWindow(){};

    void resizeEvent ( QResizeEvent * event )
    { QMessageBox* msg = new QMessageBox(this);
      msg->setText("Mainwindow has been resized!");
      msg->show();
    };

};

int main(int argc, char **argv)
```

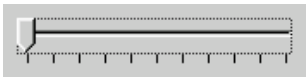
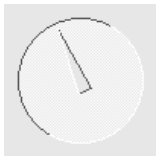
```
{
  QApplication app(argc,argv);

  myWindow widget;

  ...
   // Afficher le widget principal
  widget.show();
  return app.exec();
```

2 Utilisation de widgets et de slots existants

N'importe quel composant de la librairie Qt peut être traité comme un objet possédant des entrées (slots) et des sorties (signaux). On pourrait les comparer à des composants électroniques ou des neurones...
Prenons un exemple concret : Supposons que nous voulions connecter un objet de classe QSlider à un objet de classe QDial

QSlider :  QDial : 

La classe QSlider possède un **signal** appelé `valueChanged`. Ce signal est envoyé lorsque l'on bouge le curseur, ce qui fait varier une valeur entière (integer). La classe QDial possède un **slot** appelé `setValue`. Ce slot accepte une valeur entière (integer) comme argument et modifie la position de l'aiguille sur le cadran. Le type d'objet envoyé par le signal `valueChanged` d'un QSlider étant compatible avec le type d'objet reçu par le slot `setValue` d'un QDial, nous pouvons faire une connexion entre les deux.

1. Ecrivez les lignes de code permettant de créer ces objets des classes QSlider et QDial (instances nommées par exemple `theslider` et `thedial`), .

   ```
   QSlider theslider( ... )
   QDial thedial( ... )

   ou bien

   QSlider *theslider = new QSlider( ... );
   QDial *thedial = new QDial( ... );
   ```

2. Ecrivez le code pour connecter ces objets (affichage sur le dial de la valeur du slider)

   ```
   connect(theslider, SIGNAL(valueChanged(int)), thedial, SLOT(setValue(int)));
   ```

3. Que réalisent les lignes de code suivantes :

   ```
   QVBoxLayout *layout = new QVBoxLayout;
   layout->addWidget(theslider);
   layout->addWidget(thedial);
   setLayout(layout);
   ```

   Mise en place d'un gestionnaire de géométrie, qui aligne les widgets en colonne verticale

3 Création d'un widget composite

Soit le code suivant :

```
#include <QApplication>
#include <QFont>
#include <QLCDNumber>
#include <QPushButton>
#include <QSlider>
#include <QVBoxLayout>
#include <QWidget>


class MyWidget : public QWidget
{
public:
    MyWidget(QWidget *parent = 0);
    QPushButton *quit;
    QVBoxLayout *layout;
```

```
    QLCDNumber *lcd;
    QSlider *slider ;
};

MyWidget::MyWidget(QWidget *parent)
    : QWidget(parent)
{
    quit = new QPushButton(tr("Quit"));
    quit->setFont(QFont("Times", 18, QFont::Bold));

    lcd = new QLCDNumber(2);
    lcd->setSegmentStyle(QLCDNumber::Filled);

   slider = new QSlider(Qt::Horizontal);
    slider->setRange(0, 99);
    slider->setValue(0);

    connect(quit, SIGNAL(clicked()), qApp, SLOT(quit()));
    connect(slider, SIGNAL(valueChanged(int)),
            lcd, SLOT(display(int)));

    layout = new QVBoxLayout;
    layout->addWidget(quit);
    layout->addWidget(lcd);
    layout->addWidget(slider);
    setLayout(layout);
}

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    MyWidget widget;
    widget.show();
    return app.exec();
}
```

1. De quelle classe est dérivé l'objet myWidget ? Dessinez l'apparence de ce nouvel Widget et détaillez la fonctionnalité de celui-ci.

   QWidget The QWidget class is the base class of all user interface objects

   Plus de 190 méthodes publiques (addAction () setFont(), setWindowIcon(), ...) et de nombreux public slots :

   ```
   bool close ()
   void hide ()
   void lower ()
   void raise ()
   void repaint ()
   void setDisabled ( bool disable )
   void setEnabled ( bool )
   void setFocus ()
   void setHidden ( bool hidden )
   void setStyleSheet ( const QString & styleSheet )
   virtual void setVisible ( bool visible )
   void setWindowModified ( bool )
   void setWindowTitle ( const QString & )
   void show ()
   void showFullScreen ()
   void showMaximized ()
   void showMinimized ()
   void showNormal ()
   void update ()
   ```

2. myWidget créé un nouveau widget composite. Quel est le conteneur qui assure le placement géométrique des widgets fils ?

   QVBoxLayout

3. Ce nouvel objet possède t'il des données membres ou des methodes qui lui sont propres ? La methode `display()` est appliqué sur quel objet ? Cette methode doit-elle être définie par l'utilisateur ?

   La methode display est un public slot de l'objet QLCDNumber (voir annexes), en conséquence, ce n'est pas la peine de l'écrire.

4. Je souhaite initialiser chaque objet de type `myWidget` avec une valeur aléatoire lors de sa création. A quel endroit dois-je utiliser la fonction `rand()`. Quels sont les lignes de code à ajouter ?

   Evidemment dans le constructeur. Voici un exemple de code pour le contructeur...

   ```
   MyWidget::MyWidget(QWidget *parent)
       : QWidget(parent)
   {
    int r;
    srand ( time(NULL) );    //initialisation de rand()
    r=rand()%99;   //choisit un nombre aléatoire entre 0 et 99 que l'on appelle r
    (modulo 99, rand() allant jusqu'à RAND\_MAX>>100).
   ...
     // après création du slider
   ```

```
  ...
  slider->setValue(r);  // positionne le slider à la valeur r
...
// apres création du lcd et connection
...
  lcd->display(r); //affiche la valeur aléatoire sur le LCD
}
```

5. Pour l'instant, nous utilisons des signaux (SIGNAL) et actions (SLOTS) déjà définis dans le widget dont myWidget hérite, ou bien déjà definis dans les widgets manipulés. On s'interesse maintenant à ajouter une nouvelle fonctionnalité : l'afficheur LCD doit passer à 0 lorsque l'on clique dessus. Cela vous semble t'il faisable (voir annexes) ? Que faut-il faire pour que cette fonctionnalité puisse être implementée en écrivant la ligne de code :

```
connect(lcd, SIGNAL(reset(int)),lcd, SLOT(display(int)));
ou bien
connect(lcd, SIGNAL(reset(int)),slider, SLOT(setValue(int)));
```

```
// Fichier MyLCD.h
#ifndef MYLCD_H
#define MYLCD_H

#endif // MYLCD_H
#include <QLCDNumber>


class MyLCD : public QLCDNumber
{
Q_OBJECT

public:
MyLCD( int, QWidget *);

slots
void MyLCD::mousePressEvent(QMouseEvent *);

signals:
void reset(int );

};
```

```
// Fichier MyLCD.cpp
#include "MyLCD.h"

MyLCD::MyLCD( int value=0, QWidget * parent = 0): QLCDNumber(value,parent)
{
    printf("Constructeur de MyLCD \n");
connect(this,SIGNAL(reset(int)),this,SLOT(display(int)));
}

void MyLCD::mousePressEvent(QMouseEvent *event) // Virtual  function inherited from QWidget.
 Must be reimplemented to handle mousePress Events (and declare as slots if needed)
{
printf("MyLCD pressé! \n");
emit reset(0);
}
```

```
#include <QApplication>
#include <QFont>
#include <QLCDNumber>
#include <QPushButton>
#include <QSlider>
#include <QVBoxLayout>
#include <QWidget>
#include<math.h>
#include "MyLCD.h"



class MyWidget : public QWidget
{
public:
    MyWidget(QWidget *parent = 0);
    QSlider *slider;
    QPushButton *quit;
    QVBoxLayout *layout;
    MyLCD *lcd;
    int r;
};
```

```
MyWidget::MyWidget(QWidget *parent)
    : QWidget(parent)
{
    quit = new QPushButton(tr("Quit"));
    quit->setFont(QFont("Arial", 20, QFont::Bold));
    lcd = new MyLCD(2,NULL);
    lcd->setSegmentStyle(QLCDNumber::Filled);

    slider = new QSlider(Qt::Horizontal);
    slider->setRange(0, 99);
    connect(quit, SIGNAL(clicked()), qApp, SLOT(quit()));
    connect(slider, SIGNAL(valueChanged(int)),
            lcd, SLOT(display(int)));
    connect(lcd, SIGNAL(reset(int)),slider,SLOT(setValue(int)));

      srand (time(NULL) );
      r =rand()%99;
    slider->setValue(r);

    layout = new QVBoxLayout;
    layout->addWidget(quit);
    layout->addWidget(lcd);
    layout->addWidget(slider);
  setLayout(layout);
}


}
```

4 Création d'une fenêtre principale avec menu

On souhaite réaliser une fenêtre principale avec un menu. Pour cela on va créer une classe `SimpleMenu` dont la déclaration est :

```
 #ifndef SIMPLEMENU_H
#define SIMPLEMENU_H

#include <QMainWindow>

class SimpleMenu : public QMainWindow
{
  public:
    SimpleMenu(QWidget *parent = 0);

};

#endif
```

1. Rappeler le principe de la création d'un menu et d'items dans une fênetre principale d'une application Qt *The QAction class provides an abstract user interface action that can be inserted into widgets. QAction permet l ?ajout de raccourcis clavier, de tooltips (messages d ?aide, et l ?utilisation de connexions signaux/slots pour declencher une action.*
   Après avoir créé les actions, ces objets abstraits sont insérés dans les widgets disponibles au sein de la classe QMainWindow, par exemple, dans une barre de menu.
2. Quelles sont les classes à inclure pour inclure un menu `File` avec un item `Quit`
3. Implémenter la classe `SimpleMenu.cpp`
4. Implémenter le fichier `Main.cpp`

Fichier SimpleMenu.cpp:

```
#include "simplemenu.h"
#include <QMenu>
#include <QMenuBar>
#include <QApplication>


SimpleMenu::SimpleMenu(QWidget *parent)
    : QMainWindow(parent)
{

  QAction *quit = new QAction("&Quit", this);

  QMenu *file;
  file = menuBar()->addMenu("&File"); //menuBar() est membre de QMainWindow
  file->addAction(quit);

  connect(quit, SIGNAL(triggered()), qApp, SLOT(quit()));
```

```
}
```

Fichier Main.cpp:

```
#include "simplemenu.h"
#include <QApplication>


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

  SimpleMenu window;

  window.setWindowTitle("Simple menu");
  window.show();

  return app.exec();
}
```

Annexes

Classe QGridLayout et méthodes de QGridLayout The QGridLayout class lays out widgets in a grid. QGridLayout takes the space made available to it (by its parent layout or by the parentWidget()), divides it up into rows and columns, and puts each widget it manages into the correct cell.
Columns and rows behave identically.
QGridLayout also includes two margin widths : the border and the spacing. The border is the width of the reserved space along each of the QGridLayout's four sides. The spacing is the width of the automatically allocated spacing between neighboring boxes.

```
QGridLayout::QGridLayout ( QWidget * parent )
```

Constructs a new QGridLayout with parent widget, parent. The layout has one row and one column initially, and will expand when new items are inserted.

```
void QGridLayout::addWidget ( QWidget * widget, int row, int column,
Qt::Alignment alignment = 0 )
```

Adds the given widget to the cell grid at row, column. The top-left position is (0, 0) by default.
The alignment is specified by alignment. The default alignment is 0, which means that the widget fills the entire cell.

```
void QGridLayout::setRowStretch ( int row, int stretch )
```

Sets the stretch factor of row row to stretch. The first row is number 0.
The stretch factor is relative to the other rows in this grid. Rows with a higher stretch factor take more of the available space. The default stretch factor is 0. If the stretch factor is 0 and no other row in this table can grow at all, the row may still grow.

```
void QGridLayout::setColumnStretch ( int column, int stretch )
```

Sets the stretch factor of column column to stretch. The first column is number 0. The stretch factor is relative to the other columns in this grid. Columns with a higher stretch factor take more of the available space.
The default stretch factor is 0. If the stretch factor is 0 and no other column in this table can grow at all, the column may still grow.

Class QLCDNumber The QLCDNumber widget displays a number with LCD-like digits.
Public Slots :

```
void display ( const QString & s )
void display ( int num )
void display ( double num )
virtual void setHexMode ()
virtual void setDecMode ()
virtual void setOctMode ()
virtual void setBinMode ()
virtual void setSmallDecimalPoint ( bool )
```

Class QMenuBar

```
QAction * QMenuBar::addAction ( const QString & text )
This convenience function creates a new action with text.
The function adds the newly created action to the menu's list of actions, and returns it.

QAction * QMenuBar::addAction ( const QString & text, const QObject * receiver, const char * member )
This convenience function creates a new action with the given text.
The action's triggered() signal is connected to the receiver's member slot.
The function adds the newly created action to the menu's list of actions and returns it.
```

QApplication QApplication -> QCoreApplication.
The QCoreApplication class provides an event loop for console Qt application

```
Public Slots

void quit ()
```