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

## TD 4 - *Programmation Qt*

© B. Besserer, R. Péteri

Année universitaire 2016-2017

## 1 Pour remplacer le "printf"...

Il peut être utile à tout moment de disposer d'une zone de sortie de texte, afin d'afficher divers messages lors d'un traitement. On vous propose d'étudier 2 solutions.

### 1.1 Barre d'état

Si votre fenêtre d'affichage hérite de QMainWindow, une barre d'état est à votre disposition. Cette barre d'état permet l'affichage de 3 catégories de messages :

**Temporary** *briefly occupies most of the status bar. Used to explain tool tip texts or menu entries, for example.*

**Normal** *occupies part of the status bar and may be hidden by temporary messages. Used to display the page and line number in a word processor, for example.*

**Permanent** *is never hidden. Used for important mode indications, for example, some applications put a Caps Lock indicator in the status bar.*

*To display a temporary message, call showMessage() (perhaps by connecting a suitable signal to it). To remove a temporary message, call clearMessage(), or set a time limit when calling showMessage() :*

```
statusBar()->showMessage("Done.", 2000); // Final message for 2 seconds
```

Le mécanisme SIGNAL/SLOT peut être utilisé pour passer des messages à la barre d'état.

**Remarque :** La barre d'état n'est pas forcement visible lors du lancement de l'application. Elle apparaitra lorsqu'on y fera référence.
Soit le code ci-dessous, qui est censé simuler un processus de lecture ou de récupération de données depuis le réseau :

```
class myLoadingProcess : public QObject
{
  Q_OBJECT
  public:
    myLoadingProcess();  // constructeur
    ~myLoadingProcess(); // destructeur
  protected slots:
    void gettimeout();
  private:
    int m_progress;  // variable illustrant l'avancement de la tâche, en %
    QTimer *m_timer;
};

myLoadingProcess::myLoadingProcess()  // constructeur
{
  m_progress = 0;
  m_timer = new QTimer(this);  // création d'un timer
  connect(m_timer, SIGNAL(timeout()), this, SLOT(gettimeout()));
  m_timer->start(1000); // timeout de 1 seconde
}

void myLoadingProcess::gettimeout()
```

```
{
m_progress += 10;
  if(m_progress > 100)
  {
    m_timer->stop();
    return;
  }
  // creation d'un message texte
  QString message("avancement ");
  message = message + QString::number(m_progress) + "%";
}
```

Dans le programme principal, lorsqu'on va par exemple choisir une action de type "open" ou "download", on aura le code suivant :

```
void myMainWindow::open()
{
  myLoadingProcess *loader = new myLoadingProcess;  // creation d'un processus de chargement
}
```

Et un affichage de type avancement X% doit apparaitre dans la barre d'état.

1. De quelle manière les "impulsions" régulières du Timer sont elles prises en compte par la classe ?
   *En connectant le signal `timeout()` envoyé régulièrement par le timer (valeur passée en paramètre lors du start) à ma méthode `myLoadingProcess::gettimeout()`*
   *`connect(m_timer, SIGNAL(timeout()), this, SLOT(gettimeout()));` dans le constructeur de la classe*
2. Que contient la chaine de caractère (*QString*) `message` après la dernière ligne de la méthode `myLoadingProcess::gettimeout()` ?
   *La classe QString est une classe performante pour la manipulation des chaines de caractères. Plusieurs opérateurs sont surchargés, et '+' permet la concaténation.*
   *si `m_progress` vaut 10, et message ="avancement ", alors le code*
   ```
   message = message + QString::number(m_progress) + "%";
   ```
   *fournira la chaine "avancement 10%" c'est équivalent à*
   ```
   sprintf(message, "avancement %d\%", m_progress);
   ```
3. On souhaite récupérer le message d'avancement au niveau d'une méthode de notre classe de fenêtre principale. Nous allons donc utiliser la technique SIGNAL/SLOT :
   (a) Dans la méthode `myLoadingProcess::gettimeout()`, ajoutez l'envoi d'un SIGNAL, qui passera en paramètre la chaîne de caractère `message` créé. Ce signal doit évidemment apparaitre dans la définition de classe (fichier.h)
   *CORRIGE pour le fichier de definition .h*
   ```
   class myLoadingProcess : public QObject
   {
     Q_OBJECT
     ...
     signals:
       void reportprogress(QString message);
     ...
   };
   ```
   *CORRIGE pour le code .cpp*
   ```
   void myLoadingProcess::gettimeout()
   {
     m_progress += 10;
     if(m_progress > 100)
     {
       m_timer->stop();
       return;
     }
     // creation d'un message texte
     QString message("avancement ");
     message = message + QString::number(m_progress) + "%";
     // envoi du message en tant que SIGNAL
     emit reportprogress(message);
   }
   ```
   (b) Ajouter le code nécessaire à la méthode myMainWindow : :open() pour connecter ce signal au slot d'affichage de la barre d'état (showMessage peut être utilisé comme méthode SLOT).

```
void myMainWindow::open()
{
  myLoadingProcess *loader = new myLoadingProcess;
  connect(loader, SIGNAL(reportprogress(QString)),statusBar(), SLOT(showMessage(QString)));
}
```

## 1.2 Zone d'affichage statique

Une autre solution pour afficher des messages consiste simplement a ajouter un objet de type QLabel quelque part sur votre interface graphique, et de conserver le pointeur vers cet objet en tant que variable membre de votre classe de type fenêtre principale.

1. Quelle sont les fonctions de ces lignes dans le constructeur (on suppose que le gestionnaire de géométrie affiche mon `QLabel` au bon endroit)

```
myMainWindow::myMainWindow()
{
  ....
  m_infoLabel = new QLabel(tr("Zone d'affichage, peut toujours servir"));
  m_infoLabel->setFrameStyle(QFrame::StyledPanel | QFrame::Sunken);
  m_infoLabel->setAlignment(Qt::AlignCenter);

  QVBoxLayout *vbox = new QVBoxLayout;
  vbox->addWidget(m_infoLabel);
  ....
}
```

2. Soit les lignes de code suivant. Quel résultat obtient-on ?

```
void myMainWindow::open()
{
  m_infoLabel->setText(tr("Invoked <b>File|Open</b>"));
}

void myMainWindow::save()
{
  m_infoLabel->setText(tr("Invoked <b>File|Save</b>"));
}
```

*Affichage lorsqu'on selectionne l'article "open" du menu "file" :*


# 2 Tracé graphique avec Qt

On souhaite créer un widget dont la tâche principale sera l'affichage graphique. On souhaite par exemple le fonctionnement suivant :

— Dans ce widget, dessiner un carré de 10 x 10 pixels centré à l'endroit du clic.
— Cette zone de dessin doit bien sûr être rafraichie, lorsque, par exemple, la fenêtre que l'application était recouverte et redevient visible.

On propose la définition suivante pour cette classe (`myDrawArea`)

```
class myDrawArea : public QWidget
{
  Q_OBJECT
  public:
    myDrawArea();  // constructeur
    ~myDrawArea(); // destructeur
  protected: // reimplement virtual function
    void paintEvent(QPaintEvent *event);
    void mousePressEvent(QMouseEvent *event);
 };
```

**Important :** In Qt 4 it is not possible to draw on widgets outside of paint events.

1. Quelle méthode faut-il surcharger pour récupérer le clic (et les coordonnées du clic) ?
   *mousePressEvent. On recupère la position en invoquant la methode pos() qui retourne un QPoint, par exemple :* `QPoint myPos = event->pos();`
2. Quelle méthode faut-il surcharger pour effectuer le tracé
   *paintEvent*
3. Il faut donc mémoriser la position du rectangle pour l'afficher, car le dessin n'est possible qu'en réaction d'un événement de type paintEvent. Que proposez-vous ? Complétez le code ci-après. Voir annexes pour les méthodes nécessaire. Qu'effectue la méthode `update()` ?

```
void myDrawArea::paintEvent(QPaintEvent *)
{
  QPainter painter(this);
  ... //  a completer
}

void myDrawArea::mousePressEvent(QMouseEvent *event)
{
  ... // a completer
  update();
}
```

*CORRIGE : La solution consiste à déclarer une variable membre de la classe myDrawArea. Cette variable membre recevra une valeur lors du clic suivi de l'appel de la méthode update() qui va générer un paintEvent. La méthode paintEvent récupère la valeur de la variable membre et affiche le carré. Cette variable membre peut être du type QPoint. Ici, c'est un objet de type QRect qui a été choisi. Remarquez les méthodes que l'on peut appliquer sur un objet de type QRect ;*

```
void myDrawArea::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
  painter.fillRect(m_LastRect, Qt::red );
}

void myDrawArea::mousePressEvent(QMouseEvent *event)
{
m_LastRect.setWidth(10);
m_LastRect.setHeight(10);
m_LastRect.moveCenter(event->pos());
update();
}
```

4. Si l'on souhaite afficher les 10 derniers carrés, que proposez-vous ?
   *Memoriser les coordonnées dans un tableau, ou mieux, un tableau dynamique.*

```
#include "mywidget.h"
#include <QtGui>

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent)
{
    myRect_List = new QVector<QRect>;
}

MainWindow::~MainWindow()
{

}

void MainWindow::mousePressEvent(QMouseEvent * event)
{
    myRect_List->append(QRect(event->pos()-QPoint(5,5),event->pos()+QPoint(5,5)));
    this->update();
}

void MainWindow::paintEvent(QPaintEvent *e)

{
    QPainter myPaint(this);
    QVector<QRect>::iterator iterator;
    for(iterator = myRect_List->begin(); iterator != myRect_List->end(); iterator++)
    {
        myPaint.fillRect(*iterator,Qt::red);
    }

}
```

5. Par la suite, un contrôle supplémentaire, sous forme de spinbox (classe QSpinBox) est ajouté. Ce contrôle doit pouvoir être réglé de 1 à 10, et constitue un facteur d'échelle pour l'ensemble de vos tracé graphique. Expliquez votre demarche pour intégrer ce facteur d'echelle dans votre code ? Comptez-vous alterer les positions mémorisés ?

   *Première prise de conscience du modèle document/vue. Les points mémorisés constituent le **document**, les coordonnées sont mémorisé à l'échelle 1 (généralement le système de coordonnées de l'écran). On y ajoute la taille par défaut des carrés, et on ne modifie pas le **document**. La représentation graphique constitue la **vue**. Le plus simple et de calculer la vue à chaque affichage :*
   - *soit le facteur d'échelle (m_scale) est une donnée membre du widget d'affichage, et à chaque modification du spinbox, la donnée membre est réactualisée. On altère la position des points et la taille des carrés lors du tracé en multipliant par exemple par m_scale*
   - *soit on interroge directement la spinbox lors du tracé. Au lieu de la donnée membre m_scale, il faut alors mémoriser le pointeur vers l'objet de type spinbox (QSpinBox *m_scalespinbox), et lors du tracé, on utilise `m_scalespinbox->value()`*

# Annexes

### Methode QMenuBar : :addAction

```
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.
```

### void QWidget : :mousePressEvent ( QMouseEvent * e ) [virtual protected]

```
This event handler, for event e, can be reimplemented in a subclass to receive mouse press events for the widget.

The QMouseEvent class contains parameters that describe a mouse event.

Qt automatically grabs the mouse when a mouse button is pressed inside a widget;
the widget will continue to receive mouse events until the last mouse button is released.

A mouse event contains a special accept flag that indicates whether the receiver wants the event.
You should call ignore() if the mouse event is not handled by your widget.
A mouse event is propagated up the parent widget chain until a widget accepts it with accept(),
or an event filter consumes it.
```

### void QWidget : :paintEvent ( QPaintEvent * event ) [virtual protected]

```
This event handler can be reimplemented in a subclass to receive paint events
which are passed in the event parameter.

A paint event is a request to repaint all or part of the widget.
It can happen as a result of repaint() or update(),
or because the widget was obscured and has now been uncovered, or for many other reasons.
```

### const QPoint & QMouseEvent : :pos () const

```
Returns the position of the mouse cursor, relative to the widget that received the event.
```

### Classe QPoint

```
The QPoint class defines a point in the plane.

A point is specified by an x coordinate and a y coordinate.
The coordinates are specified using integer numbers.

The coordinates are accessed by the functions x() and y();
they can be set by setX() and setY() or by the reference functions rx() and ry().

Given a point p, the following statements are all equivalent:

    p.setX(p.x() + 1);
    p += QPoint(1, 0);
    p.rx()++;

A QPoint can also be used as a vector.
Addition and subtraction of QPoints are defined as for vectors
(each component is added separately).
You can divide or multiply a QPoint by an int or a qreal
```

### void QRect : :setWidth(int width)

```
Sets the width of the rectangle to the given width. The right edge is changed, but not the left one.
```

### void QRect : :setHeight(int height)

```
Sets the height of the rectangle to the given height. The bottom edge is changed, but not the top one.
```

### void QRect : :moveCenter(const QPoint & position)

```
Moves the rectangle, leaving the center point at the given position. The rectangle's size is unchanged.
```

### void QPainter : :fillRect ( const QRectF & rectangle, const QBrush & brush )

```
Fills the given rectangle with the given brush.

Alternatively, you can specify a QColor instead of a QBrush;
the QBrush constructor (taking a QColor argument) will automatically create a solid pattern brush.
```