Qt - Module IHM
TP n°3

par Thierry Vaira © v.1.00

Sommaire

Il est fortement conseillé d'utiliser la documentation Qt de référence en français (http://qt.developpez.com/doc/) ou en anglais (http://qt-project.org/doc/).

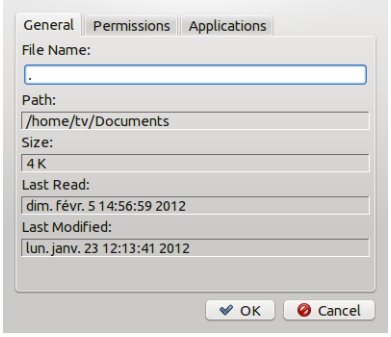
Qt

Notion de fenêtre

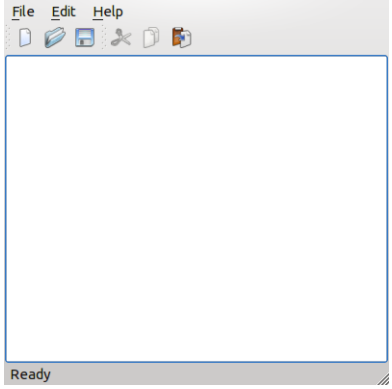
Rappel : Un *widget* qui n'est pas incorporé dans un *widget* parent est appelé une fenêtre.

Habituellement, les fenêtres ont un cadre et une barre de titre, mais il est également possible de créer des fenêtres sans décoration en utilisant des propriétés spécifiques (*window flags*).

Dans Qt, `QMainWindow` et les différentes sousclasses de `QDialog` sont les types de fenêtres les plus courantes.



Une boîte de dialogue (`QDialog`)



Une application principale (`QMainWindow`)

Boîte de dialogue (`QDialog`)

La classe `QDialog` est la classe de base des fenêtres de dialogue. Elle hérite de `QWidget`.

Une fenêtre de dialogue (ou boîte de dialogue) est principalement utilisée pour des tâches de courte durée et de brèves communications avec l'utilisateur.

Une fenêtre de dialogue (ou boîte de dialogue) :

- peut être modale ou non modale.
- peut fournir une valeur de retour
- peut avoir des boutons par défaut
- peut aussi avoir un `QSizeGrip` (une poignée de redimensionnement) dans le coin inférieur droit

Une **boîte de dialogue non modale** (*modeless dialog*) est un dialogue qui fonctionne indépendamment des autres fenêtres de la même application (Exemple : rechercher du texte dans les traitements de texte).

Une boîte de dialogue non modale est affichée en utilisant `show()` qui retourne le contrôle à l'appelant immédiatement.

Si la boîte de dialogue est visuellement cachée, il suffira d'appeler successivement `show()`, `raise()` et `activateWindow()` pour la replacer sur le dessus de la pile.

Une **boîte de dialogue modale** (*modal dialog*) est un dialogue qui bloque l'entrée à d'autres fenêtres visibles de la même application (Exemple : les dialogues qui sont utilisés pour demander un nom de fichier).

La façon la plus commune pour afficher une boîte de dialogue modale est de faire appel à sa fonction `exec()`. Lorsque l'utilisateur ferme la boîte de dialogue, `exec()` fournira une valeur de retour utile.

Une alternative est d'appeler `setModal(true)` ou `setWindowModality()`, puis `show()`. Contrairement à `exec()`, `show()` retourne le contrôle à l'appelant immédiatement (voir `QProgressDialog`).

Qt fournit un certain nombre de boîte de dialogue prêtes à l'emploi :

— La classe `QInputDialog` fournit un dialogue simple pour obtenir une valeur unique de l'utilisateur. La valeur d'entrée peut être une chaîne, un numéro ou un élément d'une liste (`getText`, `getInt`, `getDouble`, `getItem`). Une étiquette (Label) doit être placée afin de préciser à l'utilisateur ce qu'il doit entrer.

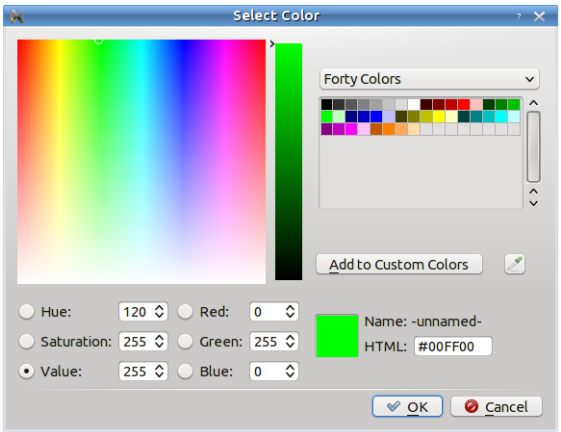
— La classe `QColorDialog` fournit un dialogue pour la spécification des couleurs. Cela permet aux utilisateurs de choisir les couleurs (`getColor`). Par exemple, vous pourriez l'utiliser dans un programme de dessin pour permettre à l'utilisateur de définir la couleur du pinceau.

— La classe `QFontDialog` fournit un widget de dialogue de sélection d'une police (`getFont`).

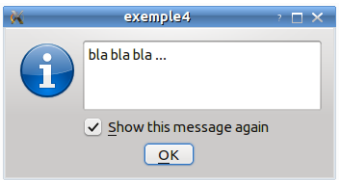
— La classe `QFileDialog` fournit une boîte de dialogue qui permet aux utilisateurs de sélectionner des fichiers ou des répertoires. Elle permet de parcourir le système de fichiers afin de sélectionner un ou plusieurs fichiers ou un répertoire (`getExistingDirectory`, `getOpenFileName`, `getOpenFileNames`, `getSaveFileName`).

— La classe `QMessageBox` fournit un dialogue modal pour informer l'utilisateur ou pour demander à l'utilisateur une question et recevoir une réponse. Elle fournit aussi quatre types prédéfinis : `QMessageBox::critical()`, `QMessageBox::information()`, `QMessageBox::question()`, `QMessageBox::warning()`

— La classe `QErrorMessage` fournit une boîte de dialogue qui affiche un message d'erreur (`showMessage()`).



QFontDialog

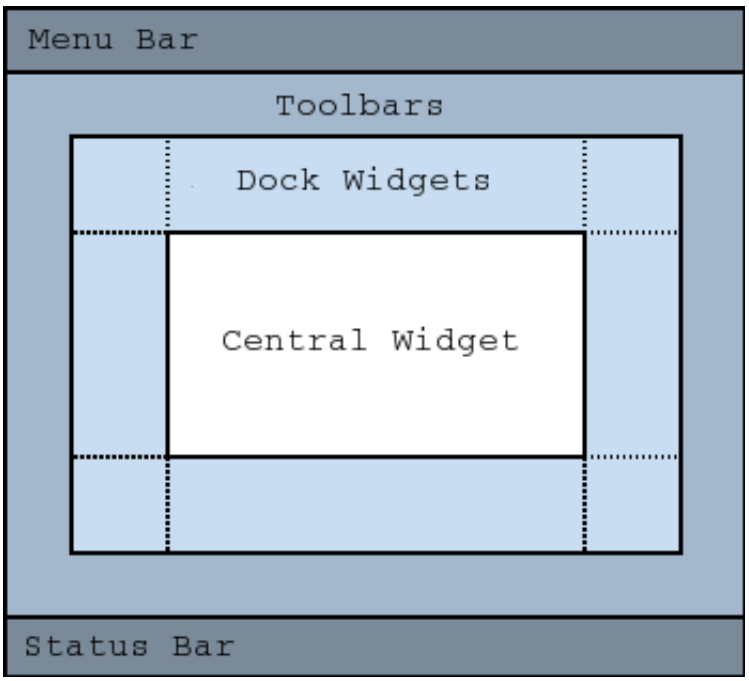


QMessageBox::information()

Fenêtre principale (`QMainWindow`)

La classe `QMainWindow` offre une fenêtre d'application principale.

Une fenêtre principale fournit un cadre pour la construction de l'interface utilisateur d'une application.



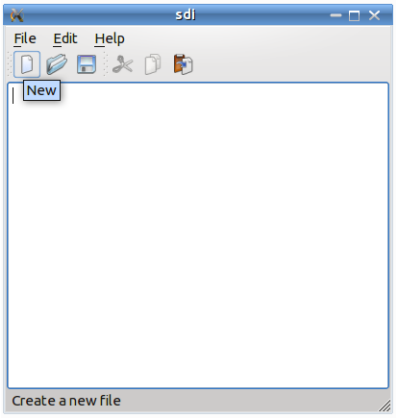
`QMainWindow` a sa propre mise en page à laquelle vous pouvez ajouter `QToolBars` (barre d'outils), `QDockWidgets` (fenêtres dockable), un `QMenuBar` (barre de menu), et un `QStatusBar` (barre d'état).

Le tracé a une zone centrale qui peut être occupée par n'importe quel type de *widget*.

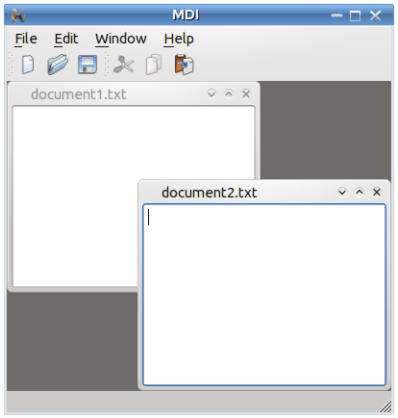
Le ***widget* central** sera généralement un *widget* standard de Qt comme un `QTextEdit` ou un `QGraphicsView`. Les *widgets* personnalisés peuvent également être utilisés pour des applications avancées.

On définit le ***widget* central** avec `setCentralWidget()`.

La fenêtre principale a soit une interface unique (**SDI** pour *Single Document Interface*) ou multiples (**MDI** pour *Multiple Document Interface*). Pour créer des applications MDI dans Qt, on utilisera un `QMdiArea` comme *widget* central.



*Application **SDI** (Single Document Interface)*



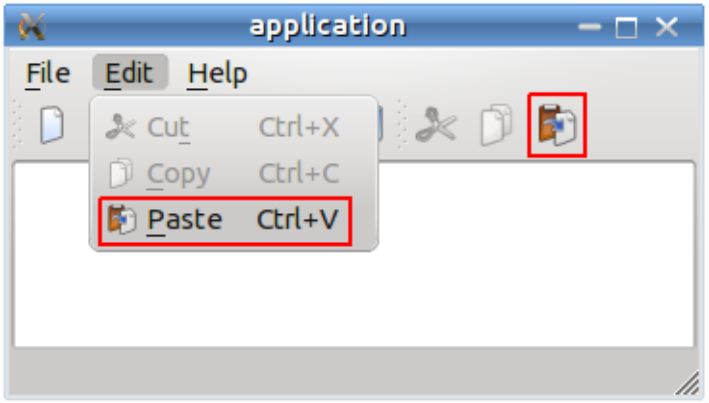
*Application **MDI** (Multiple Document Interface)*

La classe `QAction`

La classe `QAction` fournit une interface abstraite pour décrire une action (= commande) qui peut être insérée dans les *widgets*.

Dans de nombreuses applications, des commandes communes peuvent être invoquées via des menus, boutons, et des raccourcis clavier. Puisque l'utilisateur s'attend à ce que chaque commande soit exécutée de la même manière, indépendamment de l'interface utilisateur utilisée, il est utile de représenter chaque commande comme une action.

Les actions peuvent être ajoutés aux menus et barres d'outils, et seront automatiquement synchronisées.



On peut soit créer une instance de `QAction` puis l'ajouter avec `addAction()` soit créer la `QAction` directement en utilisant `addAction()`.

Les classes `QMenu` et `QMenuBar`

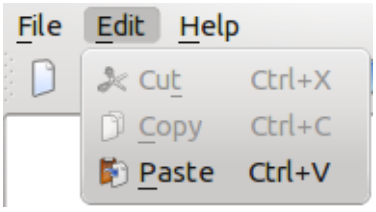
La classe `QMenu` fournit un *widget* pour une utilisation dans les barres de menus et les menus contextuels.

Un menu contextuel est un menu qui s'affiche lorsqu'on fait un clic droit sur un *widget*.

Un *widget* menu est un menu de sélection. Il peut être soit un menu déroulant dans une barre de menu ou un menu contextuel autonome. Les menus déroulants sont indiquées par la barre de menu lorsque l'utilisateur clique sur l'élément concerné ou appuie sur la touche de raccourci spécifié.

Qt implémente donc les menus avec `QMenu` et `QMainWindow` les garde dans un `QMenuBar`. On utilise `QMenuBar::addMenu()` pour insérer un menu dans une barre de menu.

La classe `QMenuBar` fournit une barre de menu horizontale. Une barre de menu se compose d'une liste d'éléments de menu déroulant.



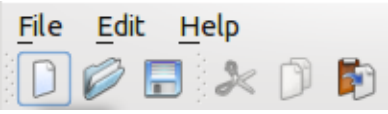
On peut ajouter un `QMenu` à un `QMenu` avec `addMenu()` pour créer un sous-menu. Chaque action d'un menu ou d'une barre d'outils peut avoir une icône `QIcon`.

La classe `QToolBar`

La classe `QToolBar` fournit une barre d'outils qui contient un ensemble de contrôles (généralement des icônes) et située sous les menus. Pour ajouter une barre d'outils, on doit tout d'abord appeler la méthode `addToolBar()` de `QMainWindow`.

Avec Qt, la barre d'outils utilise des actions pour construire chacun des éléments de celle-ci. Les boutons de la barre d'outils sont donc insérés en ajoutant des actions et en utilisant `addAction()` ou `insertAction()`.

Les boutons peuvent être séparés en utilisant `addSeparator()` ou `insertSeparator()`. Mais on peut aussi insérer un widget (comme `QSpinBox`, `QDoubleSpinBox` ou `QComboBox`) à l'aide de `addWidget()` ou `insertWidget()`.



Quand un bouton de la barre est enfoncée, il émet le *signal* `actionTriggered()`.

La classe `QDockWidget`

La classe `QDockWidget` fournit un *widget* qui peut être ancré dans une `QMainWindow` ou "flotter" comme une fenêtre de haut niveau sur le bureau.

`QDockWidget` fournit le concept de *dock windows* (palettes d'outils ou de fenêtres d'utilité). Ces *dock windows* sont des fenêtres secondaires (ou minifenêtres) placés dans la zone autour du *widget* central d'une `QMainWindow`.

Beaucoup d'applications connues les utilisent : Qt Designer, OpenOffice, Photoshop, Gimp, ...

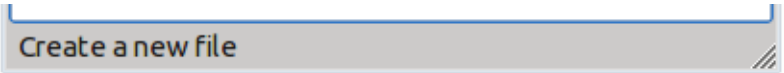
La classe `QStatusBar`

La classe `QStatusBar` fournit une barre horizontale appropriée pour la présentation des informations d'état. `QStatusBar` permet d'afficher différents types d'indicateurs.

Une barre d'état peut afficher trois types de messages différents :

- **temporaire** : affiché brièvement. Exemple : utilisé pour afficher les textes explicatifs de la barre d'outils ou des entrées de menu.
- **normal** : affiché tout le temps, sauf quand un message temporaire est affiché. Exemple : utilisé pour afficher la page et le numéro de ligne dans un traitement de texte.
- **permanent** : jamais caché. Exemple : utilisé pour des indications de mode important comme le verrouillage des majuscules.

La barre d'état peut être récupéré à l'aide de `QMainWindow::statusBar()` et remplacé à l'aide de `QMainWindow::setStatusBar()`.



On utilisez `showMessage()` pour afficher un message temporaire. Pour supprimer un message temporaire, il faut appeler `clearMessage()`, ou fixer une limite de temps lors de l'appel `showMessage()`.

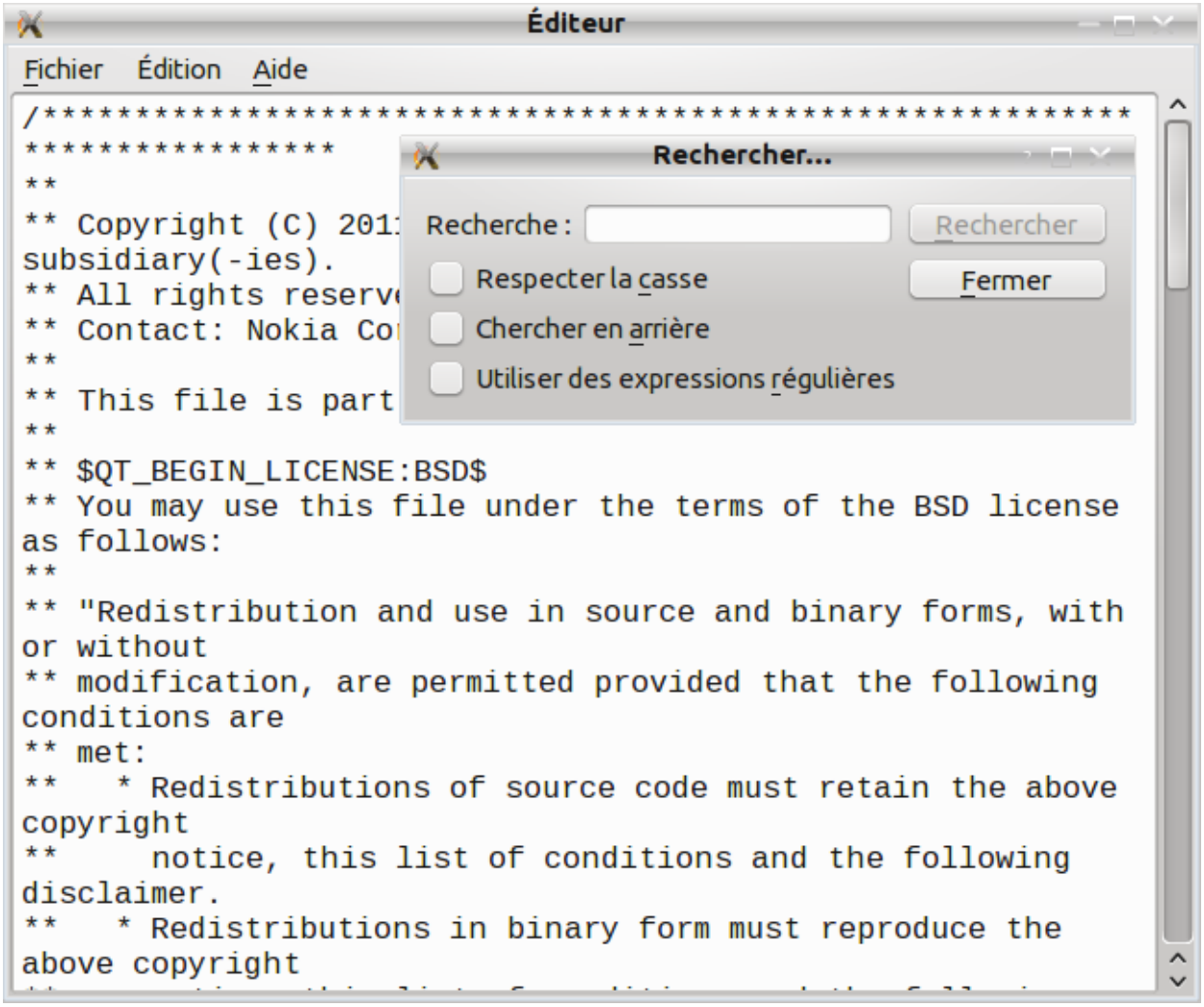
Travail demandé

Objectifs

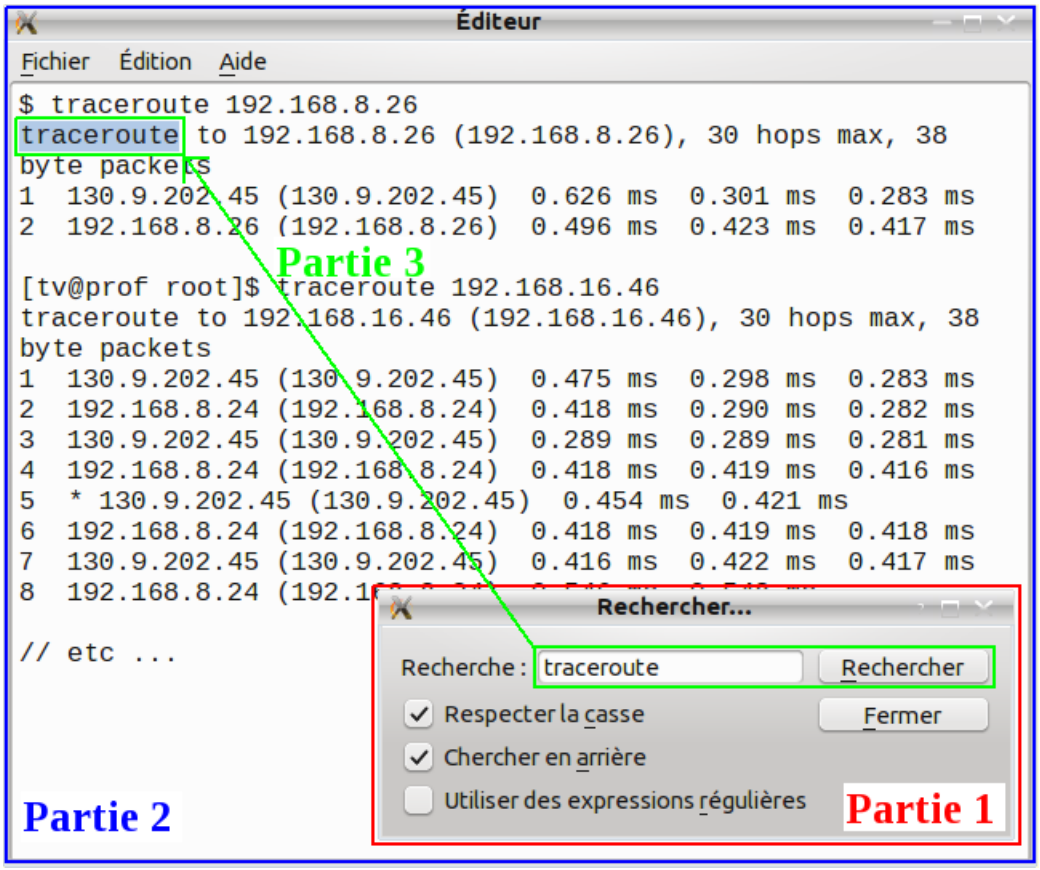
Les objectifs de ce TP sont :

- la création d'une boîte de dialogue « Rechercher ... »
- l'utilisation de cette fonctionnalité au sein d'une application principale (`QMainWindow`) de manipulation de texte (type éditeur de texte)
- la gestion de menu au sein d'une application principale (`QMainWindow`)

Ce qui donnera :



On va développer l'application en trois itérations.



Télécharger le code source fourni : http://tvaira.free.fr/info1/src-mainwindow.zip

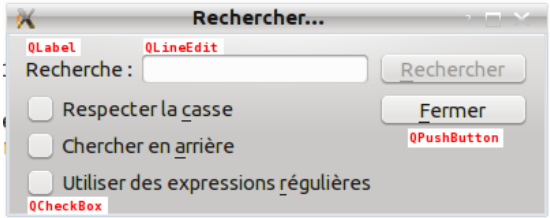
Itération 1 : la boîte de dialogue Rechercher

Cette étape est destinée à créer une boîte de dialogue « Rechercher ... ».

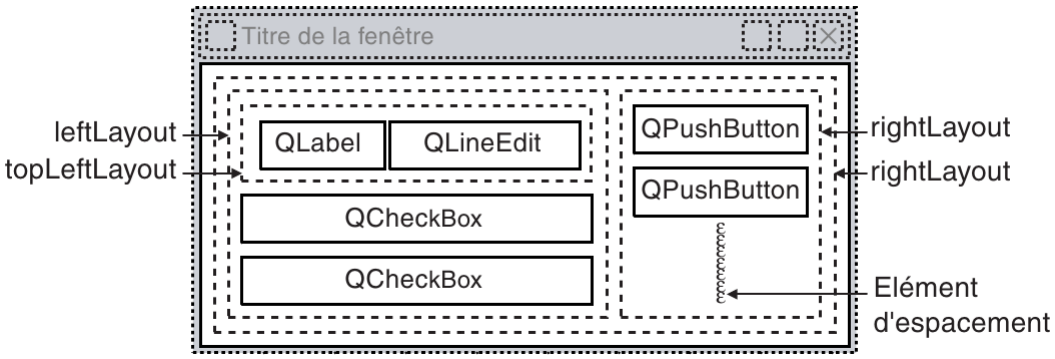
Pour cela on a créé une classe `FindDialog` qui **hérite** de `QDialog`. Le code source fourni de cette classe est réparti en deux fichiers : `finddialog.h` et `finddialog.cpp`.

Cette boîte de dialogue comprend les *widgets* suivants :

- un `QLabel`
- un `QLineEdit`
- trois `QCheckBox`
- et deux `QPushButton`



Le positionnement de ces *widgets* dans la boîte de dialogue respecte le plan suivant :



La classe `FindDialog` utilise la macro `Q_OBJECT` nécessaire pour toutes les classes qui définissent des signaux et (/ou) des *slots*.

La boîte de dialogue gére **deux signaux** quand l'utilisateur clique sur le bouton « Rechercher » :

```
SI case à cocher (checkbox) « Chercher en arrière » est activée
  ALORS émission de rechercherPrecedent(...)
  SINON émission de rechercherSuivant(...)
FINSI
```

```
QString texte = texteRecherche->text(); // QLineEdit
Qt::CaseSensitivity cs = Qt::CaseInsensitive;

if (rechercheArriere->isChecked()) // QCheckBox
  emit rechercherPrecedent(texte, cs);
else
  emit rechercherSuivant(texte, cs);
```

Code 2 – Exemple d'un texte à rechercher (en ne respectant pas la casse)

Pour la case à cocher « Respecter la casse » on utilise le type énumération `Qt::CaseSensitivity` qui peut prendre les valeurs `Qt::CaseSensitive` ou `Qt::CaseInsensitive`. Le choix de respecter la casse (ou pas) sera passé en argument du signal. Il en est de même du texte à rechercher : un `QString` ou un `QRegExp` dans le cas d'une expression rationnelle (plus communément appelée régulière).

La boîte de dialogue gére deux *slots* privés :

— `validerBouton()` qui est appelé dès que le texte change dans le `QLineEdit` et qui permettra d'activer le bouton « Rechercher » si un texte a été saisi

— `clicRechercher()` qui est invoqué lorsque l'utilisateur clique sur le bouton « Rechercher » et qui permettra l'émission d'un signal en fonction des choix des cases à cocher

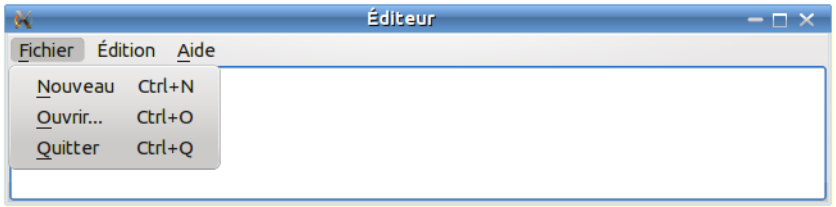
La hauteur de la boîte de dialogue est fixée à sa **hauteur optimale**. Le bouton « Rechercher » est le bouton par défaut (c'est-à-dire celui qui est pressé quand l'utilisateur appuie sur la touche Entrée). Il est aussi désactivé par défaut. La boîte de dialogue sera fermée lorsque l'utilisateur cliquera sur le bouton « Fermer ».

Itération 2 : l'application principale

Cette étape est destinée à créer la fenêtre principale de l'application en y **intégrant la boite de dialogue « Rechercher ... »**.

Pour cela on vous fournit une classe `MainWindow` qui hérite de `QMainWindow`. Le code source de cette classe est réparti en deux fichiers : `mainwindow.h` et `mainwindow.cpp`.

On a défini le menu suivant pour cette application :



Le *widget* central (`CentralWidget`) de l'application est un `QTextEdit`. On lui applique une ***Font* "Courier" taille 10**.

Cette partie de l'application est largement inspirée de l'exemple *Syntax Highlighter Example* fourni dans la documentation de Qt (http://doc.qt.nokia.com/4.7/richtext-syntaxhighlighter.html).

Vous devez compléter la classe `MainWindow` en intégrant la boîte de dialogue « Rechercher ... » réalisée à l'itération n°1. Il vous faudra donc **créer une instance** de la classe `FindDialog`. Puis lorsque l'utilisateur cliquera sur l'entrée de menu « Rechercher » (ou simplement par un Ctrl-F), on **affichera la boîte de dialogue en non modale** (*slot* `rechercher()`).

C'est l'instance de la classe `MainWindow` qui se connectera aux signaux émis par l'instance de la classe `FindDialog`. L'application principale fournira alors plusieurs *slots* publics :

— `rechercherPrecedent()` et `rechercherSuivant()` qui seront connectés aux signaux émis par la classe `FindDialog` (voir itération n°1).

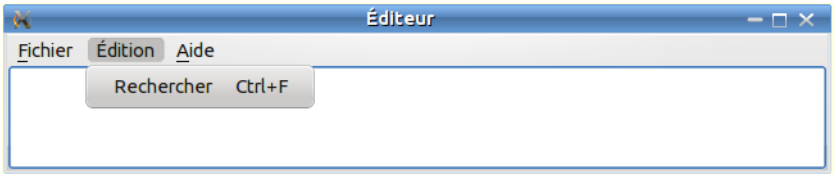
— `rechercher()` qui sera invoqué lorsque l'utilisateur cliquera sur l'entrée de menu « Rechercher » (ou Ctrl-F) et qui permet l'affichage de la boîte de dialogue de recherche.

Question 1. Compléter la définition de la méthode `setupFind()` en :

a) instanciant la boite de dialogue « Rechercher ... »

b) connectant l'ensemble des 4 signaux/*slots* (`rechercherSuivant()` et `rechercherPrecedent()`)

Question 2. Compléter la définition de la méthode `setupEditMenu()` en créant une nouvelle action avec le texte "Rechercher" et un raccourci clavier "Ctrl+F" afin d'obtenir ceci :



La nouvelle action "Rechercher" devra déclencher le *slot* membre `rechercher()`.

Itération 3 : la recherche

Cette dernière étape est destinée à utiliser la boîte de dialogue « Rechercher ... » dans l'application principale pour **effectuer des recherches**.

Le dernier travail consistera à implémenter les *slots* `rechercherPrecedent()` et `rechercherSuivant()`. Pour cela, on va utiliser le modèle `QTextDocument` associé à la vue `QTextEdit` (appel de la méthode `document()`). La classe `QTextDocument` possède plusieurs méthodes `find` à utiliser pour faire la recherche en fonction des paramètres passés par le signal (un `QString` ou un `QRegExp` pour le contenu à rechercher et un `Qt::CaseSensitivity` pour le respect de la casse ou pas).

Pour mettre en surbrillance le texte trouvé, on manipulera un `QTextCursor`.

```
void MainWindow::rechercherSuivant(const QRegExp &re, Qt::CaseSensitivity cs)
{
    QTextDocument::FindFlag fcs = (QTextDocument::FindFlag) 0;
    QTextCharFormat format;
    //format.setBackground(Qt::red);
    //format.setFontWeight(QFont::Bold);

    editor->document()->undo();
    cursor = editor->document()->find(re, cursor, fcs);
    if(cursor.position() >= 0)
    {
       //cursor.setCharFormat(format);
       editor->ensureCursorVisible();
       editor->setTextCursor(cursor);
    }
    else
    {
      // Recherche terminée !
    }
}
```

Code 3 – Le slot `rechercherSuivant()` pour une expression régulière

L'annexe n°2 fournit quelques captures d'écran à titre d'exemple.

Question 3. Compléter les définitions des méthodes `rechercherSuivant()` et `rechercherPrecedent()` pour un contenu de type `QString` à rechercher. Vous pouvez vous aider du code source fourni pour les méthodes traitant d'un contenu de type `QRegExp`.

Évidemment, une indication de fin recherche sera affichée pour prévenir l'utilisateur comme ceci :

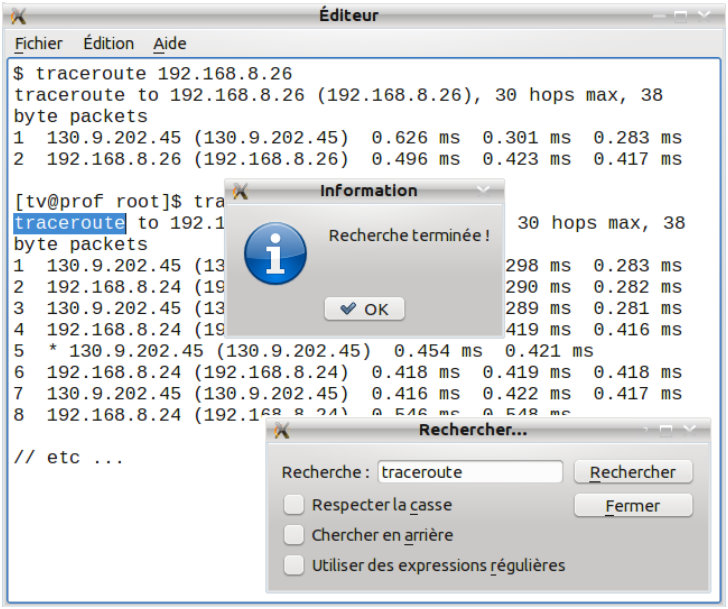


FIGURE 7 – *Affichage de l'indication de fin de recherche*

Question 4. Compléter les définitions des méthodes `rechercherSuivant()` et `rechercherPrecedent()` afin d'afficher une boîte de dialogue modale pour informer l'utilisateur que la recherche est terminée (cf. classe `QMessageBox`).

Itération 4 : bonus

Si vous avez terminé le TP, vous pouvez maintenant améliorer l'application existante.

Question 5. Ajouter au menu "Édition" les actions Couper, Copier, Coller et les rendre fonctionnelles.

Consulter la documentation sur les *slots* de la classe `QTextEdit` !

Question 6. Modifier les sources de l'application (`FindDialog` et `MainWindow`) pour assurer aussi une recherche par "mot plein" (`FindWholeWords`).

Question 7. Ajouter une barre d'outils (`QToolBar`) à l'application (`MainWindow`) pour les actions existantes.

Annexe 1 : la fonction `main()`

```
#include <QtGui>

#include "mainwindow.h"

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

    MainWindow window;
    window.resize(640, 512);
    window.show();

    return app.exec();
}
```

Code 4 – main.cpp

Annexe 2 : captures d'écran

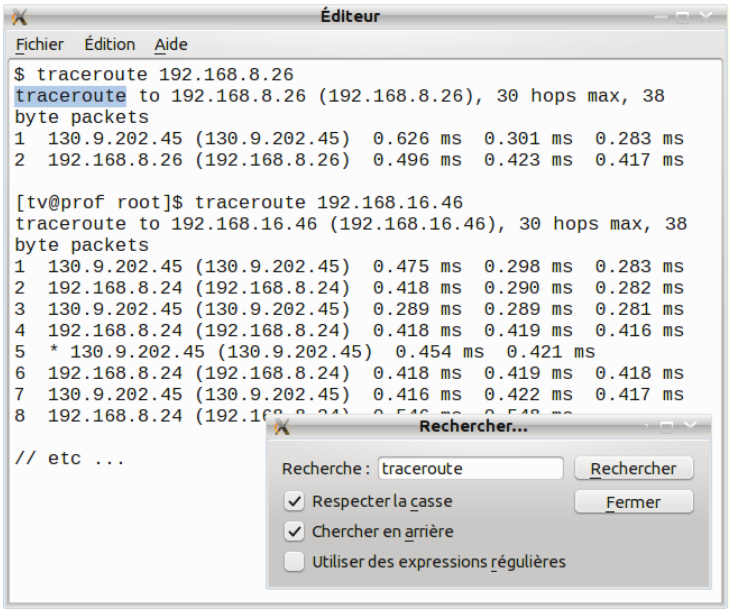


FIGURE 8 – *Une recherche arrière du mot « traceroute » en respectant la casse*

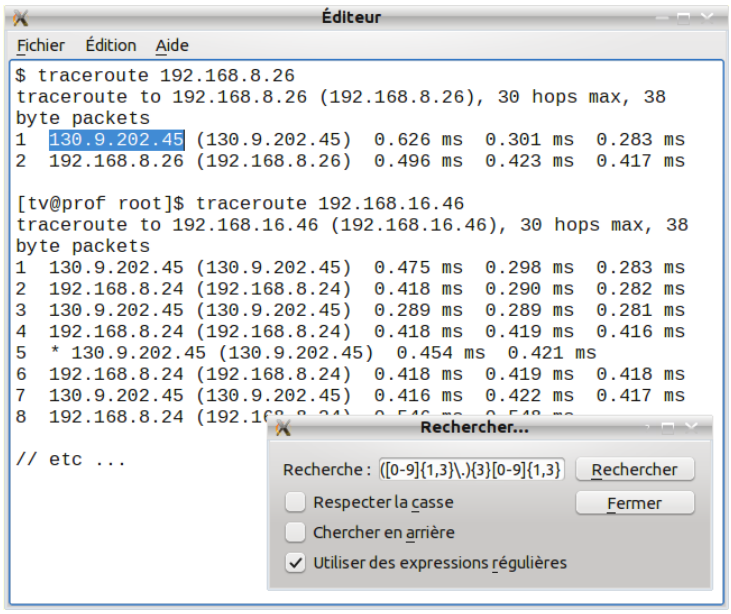


FIGURE 9 – *Une recherche à partir d'une expression régulière (ici une adresse IP)*

L'expression régulière de l'exemple permet de « *matcher* » une adresse IP en décomposant :
- `([0-9]{1,3}\.){3}` : un nombre composé de un à trois chiffres suivis d'un point, le tout répété trois fois et
- `[0-9]{1,3}` : un nombre composé de un à trois chiffres.