Création d'interfaces graphiques (GUI) avec Qt

M2I BBS

Georges Czaplicki (cgeorge@ipbs.fr), UPS / IPBS-CNRS

Table des matières (avec liens cliquables)

I. Installation de Qt sous Linux

1) La version 4.8.5 de Qt devrait être déjà préinstallée sous Fedora 19 (salle P0).

2) `qtcreator` v. 2.8.1 a été aussi installé et le chemin d'accès a été ajouté à la.variable PATH.

3) Lancer `qtdemo` pour se familiariser rapidement avec les capacités de la bibliothèque Qt.

II. Utilisation du programme `qtcreator`

1) Initialisation de travail et création de l'environnement approprié.

a) Pour commencer le travail, créer un projet vide :
- lancer `qtcreator`, sélectionner le menu "`Fichier => Nouveau projet`"
- dans le dialogue, sélectionner "`Projet Qt Widget`", "`Application graphique`"
- dans le dialogue suivant, donner un nom au projet et créer son répertoire
- dans le dialogue suivant, sélectionner la version de Qt, mode release
- le dialogue suivant permet de changer les noms des classes (ici, laissez-les intacts)
- le dernier dialogue permet optionnellement d'ajouter le nouvel objet à un projet existant
- cliquer sur "`Terminer`"

b) Cliquer sur l'icone d'écran (en bas à gauche) pour s'assurer que le bon projet et la bonne version sont pris en compte (le nom du projet et la version release, par exemple). Ceci importe pour un travail sur les projets multiples et dans le cas du *debugging*.

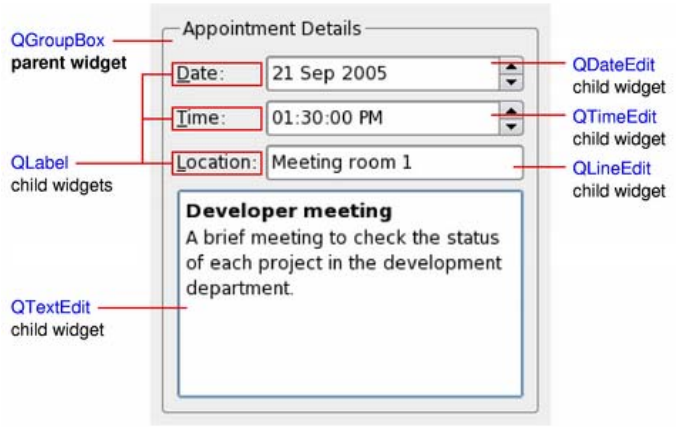
c) Cliquer sur l'icone du marteau (en bas à gauche) pour compiler le projet. On peut visualiser la sortie de compilation si on clique sur le bouton correspondant en bas de la fenêtre du `qtcreator`. Quand les boutons à gauche redeviennent verts, cliquer sur le triangle au-dessous de l'icône d'écran, ce qui lance l'exécution du programme. Une fenêtre vide apparaît. Fermez-la.

d) Il est possible de visualiser la fenêtre graphique sans compilation, en appuyant sur les boutons `Alt-Maj-R`. On peut ainsi examiner l'apparence et le comportement basique du programme, mais évidemment sans la partie codée, qui nécessite toujours la compilation. Pour sortir, appuyez sur le bouton `Echap` du clavier ou cliquez sur la croix en haut à droit de la fenêtre affichée.

Note : on peut sélectionner le style de la présentation de l'interface dans le menu `Tools => GUI Editor => Preview` où se trouvent plusieurs motifs différents, correspondant aux divers systèmes opérationnels.

2) Ajout de widgets dans la forme du Designer.

Avant de commencer, il faut se rendre compte du fait que chaque interface graphique contient des éléments multiples, souvent arrangés de façon hiérarchique. Voici un exemple qui montre les éléments constitutifs d'un fragment d'une fenêtre graphique :



Certains éléments encadrent d'autres, placés à l'intérieur, d'où le nom de parent (`parent`) pour les premiers et d'enfant (`child`) pour les derniers. Nous allons apprendre comment créer les groupes de widgets et comment les arranger sur l'interface graphique.

a) A gauche de la fenêtre du `qtcreator` se trouve l'explorateur de fichiers associés avec le projet. Double-cliquer sur `mainwindow.ui` sous `Formulaires`. `Designer` ouvre le formulaire actuel, en affichant à gauche la liste de widgets disponibles. Parcourez-la pour se familiariser avec toutes les options.

b) Sous la catégorie Boutons, placez le curseur de la souris sur le widget `Push Button`, appuyez sur le bouton gauche et – sans le relâcher – glissez l'objet sur la fenêtre en création. Relâchez le bouton de la souris quand l'objet est placé (approximativement) dans l'endroit voulu. Le bouton devient sélectionné.

c) Examiner l'éditeur de propriétés, en bas à droite de la fenêtre de `qtcreator`. Le nom du bouton (`objectName`) est `pushButton`, mais vous pouvez le changer, pour l'identifier plus facilement parmi d'autres objets qui seront ultérieurement placés sur la fenêtre graphique. Parcourez les propriétés du haut vers bas pour savoir ce qui est possible à contrôler. La propriété `text` détermine le texte affiché sur le bouton (vous pouvez aussi le changer directement en double-cliquant sur le bouton et en tapant le nouveau texte dans le champ d'entrée du bouton). La propriété `toolTip` (infobulle) permet d'afficher un texte à côté du bouton si le curseur y reste pendant environ une seconde. Tapez quelques chose dans le champ d'entrée de cette propriété (l'ellipse permet de lancer l'éditeur de texte, la flèche efface l'entrée). Appuyez sur `Alt-Maj-R` et vérifiez si l'infobulle s'affiche correctement.

3) Mise en page (*layouts*)

a) Affichez la fenêtre graphique (avec `Alt-Maj-R` ou après la compilation) et redimensionnez-la. Qu'est-ce qui se passe avec le bouton ? En fait il ne bouge pas, parce que pour l'instant on n'a rien demandé. Mais ce n'est pas l'effet désirable. Le bouton devrait suivre les changements de la fenêtre principale. Il reste de déterminer comment. Cliquez sur le fond de la fenêtre graphique et puis sur la mise en forme horizontale (le bouton avec trois lignes verticales dans le menu en haut de la fenêtre du `qtcreator`). Le bouton prend la dimension de la fenêtre. Regardez ce qui se passe avec ce bouton si vous redimensionnez la fenêtre graphique. Il suit la taille de la fenêtre, mais il occupe tout l'espace de

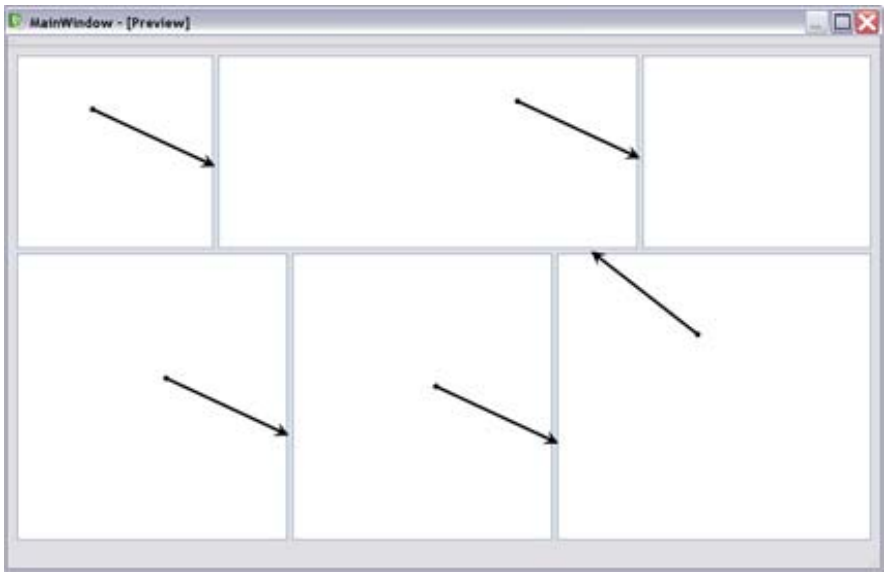
gauche à droit. Si c'est ça que vous voulez, parfait. Si non, sélectionnez le bouton et regardez les propriétés. Dans `sizePolicy` on détermine la méthode de changement de la taille de l'objet. Développez le menu (en cliquant sur le petit triangle à gauche du texte `sizePolicy`) et mettez la politique horizontale à `fixed`. Testez le comportement du bouton sur la fenêtre graphique en la lançant (`Alt-Maj-R`) et en la redimensionnant. On se rapproche un peu au comportement habituel de boutons d'une interface graphique classique.

b) Supposons qu'on souhaite voir le bouton tout le temps en bas de la fenêtre. Comment faire ? Tout d'abord cassez la mise en page précédente (avant-dernier bouton du menu au-dessus de la fenêtre du `qtcreator`). Puis, insérez un `spacer` (distanceur) vertical, en le plaçant au-dessus du bouton (il sera invisible lors d'exécution du programme). Liez les deux : sélectionnez ces deux objets et cliquez sur le layout vertical. Vous verrez un contour rouge qui entoure les deux objets. Puis, cliquez sur le fond de la fenêtre graphique pour indiquer comment placer les objets précédemment liés sur la fenêtre. Sélectionnez le `layout` horizontal. Regardez ce qui se passe en cours du redimensionnement de la fenêtre graphique.

Exercice : placez trois boutons sur la fenêtre graphique et faites en sort que le premier reste toujours en bas à gauche lors du redimensionnement, le deuxième en bas au centre et le troisième en bas à droite de la fenêtre.

c) Un autre élément intéressant et utile de l'interface graphique est le `splitter`. Il permet de changer le rapport de largeur et/ou hauteur des widgets à l'intérieur d'un espace délimité. Placez deux widgets, par exemple du type `Text Edit`, sur la fenêtre graphique, puis sélectionnez les deux et cliquez sur "`Horizontal layout in splitter`". Activez l'interface (`Alt-Maj-R`) et vérifiez comment se comportent les deux widgets "couplés" quand on déplace la barre qui les sépare. Fermez la fenêtre graphique, cassez le dernier `layout` (ou tapez `Ctrl-Z`) et refaites la procédure, mais maintenant avec "`Vertical layout in splitter`". Testez de nouveau le comportement de deux widgets.

Exercice : placez six widgets (`Text Edit`) sur la fenêtre graphique, arrangés horizontalement en deux rangées. Puis, expérimentez avec les `layouts with splitters` afin d'obtenir le résultat montré sur l'image ci-dessous : les limites librement ajustables (voir les flèches) entre les widgets de chaque rangée horizontale (quatre `splitters` verticaux), ainsi qu'entre les deux rangées (un seul `splitter` horizontal).



4) Création de groupes de widgets

a) Souvent, plusieurs widgets appartiennent au même groupe d'objets. On peut le visualiser en les mettant dans un conteneur dédié. Voici un exemple : placez le widget `Group Box` sur la fenêtre graphique. Augmentez sa taille initiale afin de pouvoir y mettre plusieurs autres objets.

Note : un `Group Box` devrait être entouré par un cadre (en fait, un `Group Box` est un `Frame` (cadre) avec un titre). Malheureusement dans certaines versions de Qt sous Linux il n'est pas visible. Dans ce cas, sélectionnez ce widget et faites un clic droit pour sélectionner "Modifier la feuille de style" du menu déroulant. Sous "Ajouter couleur" choisissez `Background color`, puis une teinte et `OK`. Dans la fenêtre de dialogue on verra le code qui attribuera cette teinte à l'arrière plan du widget (composantes RGB conseillées : 255,255,250). Ainsi on pourra voir le widget sur la fenêtre graphique, même si le cadre n'est pas visible. Il y a un moyen de restaurer le cadre manquant par la personnalisation de la feuille de style. On en apprendra plus dans la partie ultérieure de la présentation.

b) Placez trois widgets du type `Radio Button` à l'intérieur de `Group Box`, à gauche, en les empilant verticalement, puis trois autres du type `Check Box` à droite de `Group Box`, aussi l'un au-dessous de l'autre. Notez que, au moment où vous glissez les widgets sur la fenêtre graphique, le fond du widget actuellement sous le curseur change la couleur. Ceci permet d'associer les nouveaux widgets avec les éléments déjà existants. Vérifiez le fonctionnement (`Alt-Maj-R`) pour s'apercevoir que les `Radio Buttons` marchent de façon mutuellement exclusive (un seul est active à la fois), tandis que les `Check Boxes` peuvent être activés indépendamment. A retenir que ces widgets font maintenant partie du groupe associé avec l'objet `Group Box` et n'affectent pas le fonctionnement des widgets attribués à un autre groupe.

Exercice : créez deux objets du type Group Box et ajoutez dans chacun, come avant, six widgets (trois `Radio Buttons` et trois `Check Boxes` (on peut faire copier/coller). Utilisez le Grid Layout à l'intérieur de chacun de deux groupes pour arranger les widgets en grille. Puis, assurez-vous que le premier Group Box apparaîtra toujours dans le coin gauche en haut de la fenêtre graphique, et le deuxième dans le coin droit, en bas.

c) Explorez les widgets restants avec leurs propriétés (Combo Box, Line Edit, Text Edit, Spin Box, Combo Box, Dial, Scroll Bar, Slider, Progress Bar, Tab Widget,…). A la fin du travail, supprimer le projet actuel et créez un nouveau projet vide pour les exercices suivants.

5) Utilisation de slots et de signaux

a) Pour l'instant, si on clique sur les widgets il n'y a pas grand-chose qui se passe. C'est normal : on ne leur a attribué aucune action. Pour le remédier, essayez ce qui suit : placez un bouton en bas de la fenêtre graphique, avec le texte "OK". Dans la fenêtre d'éditeur de propriétés changez son nom à `ok_but`. Puis, placez deux autres widgets dans l'espace disponible : un `Horizontal Slider` et un `LCD Number`. Augmentez la taille de ce dernier pour mieux voir les chiffres qui vont s'y afficher. Cliquez sur le `Slider` pour le sélectionner et dans les propriétés mettez les valeurs min & max entre (0, 100). Dans le menu `Edit` cliquez sur "Edit signals and slots" (ou tapez F4 sur le clavier). Vous

entrez dans l'éditeur des signaux et de slots. Appuyez le bouton gauche de la souris sur le `Slider`, glisser le curseur sur le `LCD Number` et relâchez le bouton. Un dialogue s'ouvre, où à gauche vous avez la liste des signaux qui peuvent être générés par le `Slider` et à droite la liste des signaux qui peuvent être interceptés par le `LCD Number`. Cliquez à gauche sur `valueChanged(int)` et à droite sur `display(int)`, puis `OK`. Ceci veut dire que quand la position du `Slider` sera changée, l'interface va générer le signal du changement de la valeur (qui varie ici entre 0 et 100), avec la valeur représentée par un chiffre entier (`int`). Cette valeur sera passée au `LCD Number`, qui va l'afficher (`display`). Notez bien que la liste des arguments des signaux (ici seulement `int`) doit être identique au moment de l'envoie et de la réception. Pour compléter, cliquez sur le bouton OK et relâchez le bouton de la souris sur le fond de la fenêtre graphique. Dans le dialogue qui s'affiche cochez la case `Afficher les signaux et slots hérités de QWidget` pour obtenir la liste complète dans la partie droite. Connectez le signal `clicked()` avec `close()`. Notez qu'il n'y a pas d'arguments ici. Un click sur le bouton OK devrait donc fermer la fenêtre graphique. Essayez maintenant si tout marche comme prévu (Alt-Maj-R ou compilation). Déplacez le `slider` et observez l'affichage `LCD`. A la fin, cliquez sur `OK` pour sortir du programme.

b) Les signaux ci-dessus sont prédéfinis, mais malgré leur richesse ils sont plutôt basiques. Souvent on a besoin de fournir une fonctionnalité particulière, où un clic sur un bouton déclenche une action bien spécifique. Ceci peut être facilement accompli avec le minimum de programmation. Voici un exemple. A la fenêtre existante ajoutez un nouveau bouton, avec le texte "Click me". Entrez dans l'éditeur de signaux (F4), puis connectez le nouveau bouton avec la forme principale. Dans le dialogue qui s'affiche sélectionnez `clicked()` à gauche, puis cliquez sur `Editer` au-dessous de la liste à droite. Pour ajouter un nouveau slot, cliquez sur le signe "`+`" vert au-dessous de la liste de slots. Remplacez la ligne qui s'affiche avec le texte "`myslot()`" et confirmez (`OK`). Sélectionnez le nouveau slot dans la liste à droite et confirmez. La forme affiche maintenant toutes les connexions. Cependant, si vous compilez et exécutez le programme, vous verrez qu'il y a un problème : le clic sur le bouton "`Click me`" ne fait rien et un message d'erreur apparaît dans la console de sortie. Ceci est dû au fait que le nouveau slot n'a pas encore été proprement déclaré. `qtcreator` ne peut pas connaitre nos intentions et donc c'est à nous de le lui dire. Cliquez sur `Editer` à gauche de la fenêtre principale du `qtcreator` pour afficher la liste des fichiers composant le code de notre programme. Développez les `Headers` et double-cliquez sur `mainwindow.h`. Vous verrez la définition de la classe MainWindow, où il faut ajouter notre slot. Après les lignes "`public: explicit …`" insérez ces deux lignes :

```
public slots:
        void myslot();
```

Maintenant le slot est déclaré mais la compilation échoue, pour la simple raison que la sous-routine `myslots()` n'apparaît nulle part dans le code. Il faut la créer. Développez `Sources` à gauche et double-cliquez sur `mainwindow.cpp` (pour faire simple, à cette étape nous n'allons pas ajouter de nouveaux fichiers au projet). A la fin du code ajoutez les lignes suivantes :

```
#include <QtGui>

void MainWindow::myslot()
{
// le code sera ajouté ici
}
```

La première ligne assure la possibilité d'utiliser le contenu de la bibliothèque Qt dans notre code, le reste est un template d'une sous-routine, pour l'instant vide (les caractères "`//`" signifient une ligne de commentaire). Compilez et exécuter le code. Tout doit se bien passer, sauf qu'un clic sur le nouveau

bouton ne fait toujours rien. Pour remplir ce vide, voici un exemple d'utilisation d'une boîte de dialogue prédéfinie : remplacez la ligne de commentaire ci-dessus par celle-ci :

```
QMessageBox::information(this, "My information dialog",
          "Here is the text of the dialog box",QMessageBox::Ok);
```

Pour voir à quel degré c'est facile de travailler avec `qtcreator`, commencez de tapez "`QMes`" et vous verrez s'afficher un dialogue d'aide. Au lieu de tapez, on peut sélectionnez (avec la souris ou avec le clavier) ce qu'on cherche dans la liste. Après `QMessageBox` tapez "`::`" et vous verrez d'autres options (`information, warning, critical`...). Vous verrez aussi la liste des arguments avec leurs types, ce qui fait la programmation quasiment automatique. Pour voir la totalité d'information sur `QMessageBox`, cliquez sur `Aide` à gauche de la fenêtre du `qtcreator` et faites la recherche par rapport au mot-clé `QMessageBox`. Sélectionnez `Class Reference` où se trouvent toutes les options de l'appel de cette routine et fouillez un peu pour voir toutes les possibilités. Pour finir, compilez et exécutez le code.

Note : on peut utiliser les `HTML tags` dans les chaînes, ce qui introduit une flexibilité dans la programmation. Par exemple, au lieu de "`Here`" tapez "`<b>Here</b>`", au lieu de "`text`" tapez "`<i>text</i>`" et au lieu de "`dialog`" tapez "`<font color='red'>dialog</font>`". Maintenant dans la boîte de dialogue qui s'affiche on voit le mot **`Here`** en gras, le mot *`text`* en italique et le mot `dialog` en rouge. Pour la liste complète des options, consultez le chapitre "`Supported HTML Subset`" dans l'Aide du `qtcreator`.

Exercice : ajoutez deux nouveaux widgets sur votre fenêtre graphique : un `Line Edit` et un `Text Edit`. Remplacez le texte "`Click me`" sur le bouton existant par "`Copy to Editor`". Modifiez le contenu de la sous-routine `myslot()` de façon à ajouter le texte tapé dans `lineEdit` à la fin du texte dans le `textEdit`. Vérifiez le fonctionnement du programme : chaque clic sur le bouton "Copy to Editor" devrait ajouter le texte actuellement dans le champ de `lineEdit` à la fin du texte déjà existant dans `textEdit`.

6) Menus et actions

a) Pour enrichir encore plus notre interface graphique, nous allons y ajouter un menu avec quelques actions typiques. En haut à gauche de la fenêtre graphique on aperçoit le texte "`Type here`". Si vous le double-cliquez, on peut taper un texte dans un champ qui devient éditable. Ceci sera l'en-tête du premier objet de notre menu. Tapez, par exemple, `File`, et appuyer sur `Entrée` du clavier. Le texte "`Type here`" se déplace vers bas et on peut répéter la procédure, mais cette fois les textes apparaîtront comme objets du même menu. Tapez par exemple `Open`, et dans la ligne suivante `Save`. Pour faire les choses plus intéressantes, double-cliquez sur "`Add separator`", puis ajoutez encore `Quit`. Vérifiez l'effet final après Alt-Maj-R, cliquez sur le menu `File` et vous devriez voir les options spécifies. Evidemment, elles ne sont pas encore opérationnelles. Il faut associer les actions avec les entrées du menu. Voir ci-dessous.

b) En bas à gauche de la fenêtre du `designer` se trouve un onglet d'`Editeur d'actions`. Cliquez là-dessus pour trouvez les trois entrées du menu `File`. Double-cliquez la ligne `actionOpen`. Dans le dialogue qui s'ouvre on peut mettre ou modifier les propriétés de cette action. Cliquez sur le champ `Shortcut` (raccourci) et tapez, par exemple, `Ctrl-O` (c'est-à-dire appuyez sur `Ctrl` et en même temps sur `O`). Cette action sera désormais accessible soit par la sélection à partir du menu soit par le raccourci clavier. Confirmer (`OK`). Maintenant cliquez sur `actionOpen` encore une fois, mais avec le bouton droit de la souris. Sélectionnez l'option "`Go to Slot`". Ceci permettra d'attribuer un signal à cette action. Choisissez `triggered()`. L'affichage change pour montrer le code correspondant. Notez bien que `qtcreator` ajoute un template d'une sous-routine (`on_actionOpen_triggered()` dans le fichier `mainwindow.cpp`) où on peut ajouter notre propre code (aussi, il déclare cette action dans le fichier `mainwindow.h` comme un `private slot`). Pour l'instant le contenu est vide, mais on peut rapidement vérifier si cette action est déclenchée comme il faut. Mettez dans cette routine l'affichage d'un dialogue en employant `QMessageBox`. Compilez et lancez le programme, puis sélectionnez le menu File=>Open. Le dialogue devrait s'afficher, également quand on tape le raccourci `Ctrl-O`. Pour finir, ajoutez les slots pour les deux autres actions. Dans le cas de la dernière, `actionQuit`, tapez le code suivant dans l'endroit approprié : "`this->close();`", ce qui devrait fermer la fenêtre de l'interface graphique. Vérifiez-le en compilant et exécutant le code.

7) Copains (*buddies*) et l'ordre de widgets sur le formulaire

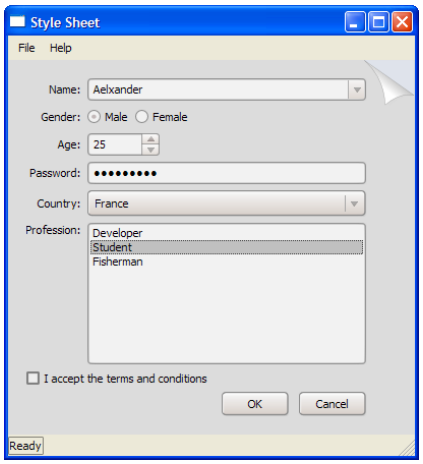
a) Il arrive souvent de voir les widgets d'entrée de données (`LineEdit`, `TextEdit`,…) associés avec les étiquettes explicatives (`Labels`). Au lieu de mettre le focus de l'interface graphique explicitement aux widgets par un clic de la souris, on peut utiliser le raccourci du clavier concernant l'étiquette associé. Comme exemple, ajoutez à la fenêtre graphique de notre exercice deux étiquettes (Labels), un `LineEdit` et un `CheckBox` (mettez la 1^{e} étiquette et `LineEdit` côte à côte, pareil pour la 2^{e} étiquette et le `CheckBox`). Double-cliquez le texte de la première étiquette et tapez "`&Prénom`", suivi par `Entrée`. Le système Qt interprète le symbole "`&`" comme indication que la lettre qui suit (ici `P`) représente le raccourci clavier (`Alt-P`) pour cette étiquette. Dans le texte la lettre `P` sera soulignée. Faites pareil pour la deuxième étiquette : tapez "`&Nom`" pour le texte, ce qui veut dire qu'elle sera accessible via le raccourci `Alt-N`. Maintenant il faut créer les pairs de copains (*buddies*). Dans le menu `Edit` du `qtcreator` sélectionnez `Edit buddies`, puis appuyez sur le bouton gauche de la souris sur la première étiquette, glissez la souris jusqu'au `LineEdit` et relâchez pour établir l'association entre les deux widgets. Répétez la procédure pour la deuxième étiquette et le `CheckBox`. Tapez `Alt-Maj-R` pour vérifier ce que ça donne. Il suffit d'appuyer sur `Alt-P` pour voir que le focus est transféré vers le `LineEdit` associé avec la première étiquette, c'est-à-dire le curseur se déplace vers le champ d'entrée de LineEdit, comme dans le cas d'un clic là-dessus. Pareil pour le deuxième pair si on tape `Alt-N`. On voit que le `CheckBox` se comporte comme après un clic si on tape le raccourci de son copain (*buddy*).

b) Pour déplacer le focus sur l'interface graphique on se sert du bouton `Tab` du clavier. Chaque fois quand on l'appuie, le focus se déplace vers le widget suivant. La question est, dans quel ordre ? On peut établir l'ordre de ces déplacements avec l'éditeur, disponible dans le menu `Edit => Tab Order`. Vous verrez les chiffres à côté des widgets, indiquant l'ordre de widgets dans la liste. En principe, pour changer l'ordre, on peut faire un click droit sur un de chiffres et sélectionner l'option de l'ordre de la liste, où on peut manuellement agir sur les éléments individuels. Mais il est plus simple de cliquer sur chacun de chiffres (parce que ceci les incrémente) jusqu'à l'apparition du chiffre désiré. Faites-le pour tous les widgets de votre interface graphique, en mettant le widget `TextEdit` à la fin de la liste. Vérifiez si tout marche bien : tapez `Alt-Maj-R` et regardez si l'objet avec le focus et celui que vous avez mis au début de la liste. Puis, appuyez `Tab` plusieurs fois pour voir si le focus se déplace de façon prévue (la raison pour laquelle on met `TextEdit` à la fin est simple : quand le focus rentre dans le

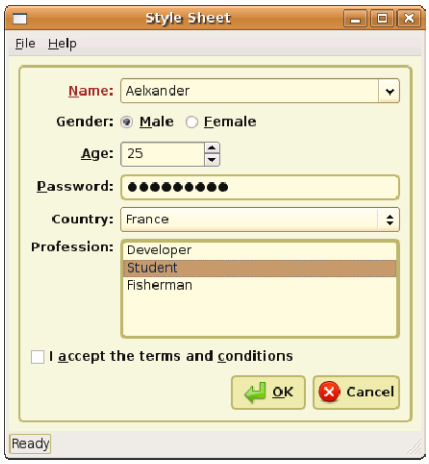
champ d'entrée, il va y rester. Chaque `Tab` est interprété comme le texte d'entrée et déplace le curseur à l'intérieur de ce widget et non vers le widget suivant).

8) Les feuilles de style

a) Le *look* de l'interface graphique et de ses éléments dépend du système opérationnel (MS Windows vs. Linux vs. Mac) et la bibliothèque Qt fourni les versions par défaut. Par exemple :



Windows XP



Linux Ubuntu

Max OS X

Cependant, on peut contrôler la quasi-totalité des paramètres définissant la nature des widgets. La meilleure méthode de se familiariser avec ces problèmes est de lire la documentation (voir, par exemple, "`Qt Style Sheets`" ainsi que "`Customizing Qt Widgets Using Style Sheets`" dans l'`Aide` du `qtcreator`). Beaucoup d'exemples de personnalisation du style de widgets divers peuvent être trouvés dans le chapitre "`Qt Style Sheets Examples`". Ici, on se limitera à une démonstration de propriétés basiques qui peuvent être contrôlées de façon facile.

On peut changer les propriétés de widgets de façon dynamique, pendant l'exécution du code, ou initialiser les widgets avec les valeurs désirées au moment du lancement du programme. Nous allons explorer cette dernière option. Pour ce faire, nous allons créer une sous-routine d'initialisation qui sera exécutée avant l'affichage de l'interface graphique. Tout d'abord, faites l'édition du fichier `main.cpp` et ajoutez la ligne "`w.init();`" juste avant "`w.show();`". Editez maintenant le fichier `mainwindow.h` en y ajoutant "`void init();`" dans les `Public slots`. Finalement, ouvrez le fichier mainwindow.cpp et ajoutez, à la fin du code, le template vide de la sous-routine `init()`. La compilation doit se terminer correctement.

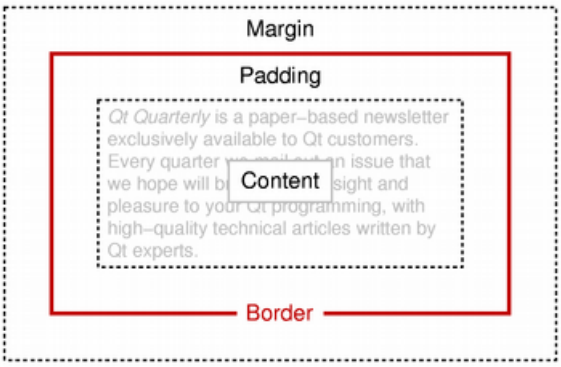
b) Commençons par la personnalisation du bouton OK. Supposons que nous souhaitons voir le texte (`OK`) rouge sur le fond jaune. Il suffit d'ajouter dans la sous-routine `init()` la ligne suivante :

```
ui->ok_but->setStyleSheet("QPushButton {background-color: yellow; color: red;}");
```

Compilez le code et exécutez-le. Le bouton OK a changé ? Tout va bien. Continuons la personnalisation. Nous allons changer la police du texte. Modifiez le contenu de `init()` de façon suivante :

```
ui->ok_but->setStyleSheet("QPushButton {background-color: yellow; color: red;"
                "font: bold 12px;}");
```

Apercevez que maintenant la définition du style s'étale sur deux lignes. Ceci ne gène en rien au compilateur, il lit le code jusqu'au signe "`;`" qui termine l'entrée. Mais pour continuité, chaque ligne doit être entourée par les guillemets. Notez aussi que, à l'intérieur des accolades, chaque option est séparée d'autres par le signe "`;`". Comme avant, vérifiez si les résultats sont au niveau de vos attentes. L'étape suivant de la personnalisation sera le changement de bordures du bouton. Avant de procéder, regardez de près les composantes d'un widget :



Dans ce qui suit, on va affecter les paramètres des éléments montrés ci-dessus. Nous allons arrondir les bordures et faire sortir le bouton du plan de la fenêtre. Tapez la ligne suivante :

```
ui->ok_but->setStyleSheet("QPushButton {background-color: yellow; color: red;"
                "font: bold 12px; border-color: grey; border-width: 2px;"
                "border-radius: 10px; border-style: outset;}");
```

Les nouvelles options signifient que la bordure sera grise, avec l'épaisseur de 2 pixels, les coins seront arrondis en utilisant les arcs dont le rayon est de 10 pixels, et le bouton sortira du plan (`outset`). Comme avant, une compilation s'impose pour vérifier si tout va bien. En principe, le look devrait être comme souhaité. Le problème, c'est que le bouton est "statique", c'est-à-dire rien ne change quand on passe le curseur là-dessus ou quand on clique là-dessus. Ceci est contraire à notre expérience avec les interfaces graphiques habituelles. Pour cette raison nous allons enrichir la définition du style, en y ajoutant les spécifications du comportement du bouton dans les cas ou (i) on passe le curseur dessus (`hover`) et (ii) quand on appuie sur le bouton de la souris là-dessus (`pressed`). Vous devriez être déjà capable de comprendre ce que les lignes suivantes définissent :

```
ui->ok_but->setStyleSheet("QPushButton {background-color: yellow; color: red;"
                "font: bold 12px; border-color: grey; border-width: 2px;"
                "border-radius: 10px; border-style: outset;}"
  // ...when mouse is over the button
                "QPushButton:hover {background-color: lightblue; color: yellow;"
                "border-style: inset;}"
  // ...when button is pressed
                "QPushButton:pressed {background-color: #008BF7; color: white;"
                "border-color: black; border-width: 1px; border-style: inset;}");
```

La propriété `inset` permet de faire "entrer" le bouton dans le plan de la fenêtre graphique. Les couleurs peuvent être spécifiées par leurs noms (un jeu restreint des options) ou par leurs codes RGB (le symbole # suivi par les codes hexadécimaux (entre `00` et `FF`) des composantes rouge, verte et bleue). N'hésitez pas à expérimenter librement avec les options diverses pour apprendre leurs effets sur l'apparence de widgets et aussi pour mieux les personnaliser à votre goût.

Voici un exemple un peu plus avancé : la création d'un bouton avec au fond un gradient de couleurs qui font l'impression de la réflexion de lumière. Analysez le contenu de cette définition et consultez la documentation (e.g. `Qt Style Sheets Exemples`) pour en savoir plus, mais surtout expérimentez pour vous familiariser avec tous les aspects de cette approche :

```
ui->ok_but->setStyleSheet("QPushButton {background: qlineargradient"
            "(x1:0.5, y1:0, x2:0.5, y2:1, stop:0 #008BF7,"
            "stop:0.2 white, stop:0.4 lightblue, stop:1 #008BF7);"
            "border-style: outset; border-width: 1px; border-radius: 10px;"
            "border-color: black; font: bold 12px; min-width: 5em; padding: 6px;}"
// ...when mouse is over the button
            "QPushButton:hover {border-style: inset; color: black;"
            "background: qlineargradient(x1:0.5, y1:0, x2:0.5, y2:1, stop:0 #008BF7,"
            "stop:0.2 white, stop:0.6 lightblue, stop:1 #0AA3F5);}"
// ...when button is pressed
            "QPushButton:pressed { background: qlineargradient"
            "(x1:0.5, y1:0, x2:0.5, y2:1, stop:0 #008BF7, stop:0.2 white,"
            "stop:0.4 #0A6DD6, stop:1 lightblue); "
            "border-style: inset; border-color: black; border-width: 1px;"
            "color: white;}");
```

Exercice : placez un nouveau widget sur la fenêtre graphique : `Progress Bar`. Connectez la sortie du `Slider` déjà existant à l'entrée de la barre de progrès. Lancez le programme et vérifiez si, en bougeant le `slider`, on arrive à contrôler la barre de progrès. Apprenez à personnalisez ce widget avec les feuilles de style, en changeant les paramètres de la bordure (`border`), position du texte affiché (`text-align`), ainsi que l'apparence de l'intérieur de ce widget (`chunk`).

Voici un exemple d'une feuille de style, reprenant quelques éléments déjà discutés ci-dessus, notamment le gradient de couleurs (`progbar` est le nom attribué au widget `Progress Bar`) :

```
ui->progbar->setStyleSheet("QProgressBar {border: 2px solid grey;"
                "border-radius: 5px; border-style: outset; text-align: center;}"
                "QProgressBar::chunk {background: qlineargradient(x1: 0, y1: 0.5,"
                "x2: 1, y2: 0.5, stop: 0 #7FFFD4, stop: 1 #FF69B4);}");
```

La page Web `en.wikipedia.org/wiki/X11_color_names` contient une liste de codes RGB de couleurs. Pour aller plus loin, consultez la documentation (`Qt Style Sheets Examples`).

c) Pour finir, un exemple de personnalisation du `Group Box`. Ceci peut être important dans la mesure où dans certaines versions de Qt le cadre ne s'affiche pas sous Linux, contrairement à la version Windows. En attendant la correction de cette bogue, on peut utiliser les feuilles de style pour remédier à cette situation. Ajoutez un widget du type `Group Box` sur la fenêtre graphique. Nommez-le `groupBox`. Remplissez-le avec, par exemple, trois `Radio Buttons`. Vérifiez si tout s'affiche correctement. Quelque soit le résultat, on peut modifier l'apparence de ce widget. Modifier le fichier `mainwindow.cpp` en y ajoutant la ligne suivante :

```
ui->groupBox->setStyleSheet("QGroupBox {font: bold 12px; border: 1px solid grey;"
                            "border-radius: 5px; margin-top: 1ex;}"
                            "QGroupBox::title {subcontrol-origin: margin;"
                            "left: 10px;  padding: 0 3px;}");
```

Vous contrôlez ici la police, la bordure et la position du titre de `groupBox`. Expérimentez avec les valeurs données ci-dessus et lisez la documentation pour en apprendre d'avantage.

9) Fichiers de ressources

a) Placez un nouveau bouton (`Push Button`) sur la fenêtre graphique. Nommez le "`icon_but`". Nous allons mettre une icône sur ce bouton. Pour faire votre choix d'image, essayez, par exemple, de faire une recherche dans les images Google, par rapport aux mots-clés : "`icon png`". Le deuxième de ces mots représente le format d'image. Après le choix de votre image préférée, sauvegardez-la dans le répertoire `images`, qu'il faudra créer juste au-dessous du répertoire racine de votre projet. Supposons que le nom du fichier contenant cette image est `icon_but.png`. En général, vous pouvez ramasser dans ce répertoire toutes les images qui seront utilisées dans votre interface graphique. Elles peuvent être utilisées comme icônes, comme fonds de fenêtres et/ou widgets, etc. Normalement, il faudrait dire au programme d'aller chercher cette image dans une location spécifique sur le disque dur, mais ceci équivaut à l'hypothèse implicite que la localisation de ces fichiers ne changera jamais, sinon le programme ne les trouvera plus. On peut faire mieux, justement en créant un fichier de ressources qui permettra d'inclure toutes les images directement dans le code au moment de compilation.

Cliquez sur le menu `Edit => New` pour sélectionner un nouveau fichier Qt / fichier de ressources Qt. Donnez lui un nom, et permettez l'ajout au projet actuel. Notez que le fichier *.pro (le premier sur la liste de fichiers du projet) contient maintenant la définition de ce nouveau fichier. Faites un clic droit sur le nom du fichier de ressources et sélectionnez `Ouvrir avec => Ressource Editor`. En bas de la fenêtre du `qtcreator` cliquez sur `Ajouter => Ajouter un préfixe`. Dans le champ `Prefix` tapez "`/`" pour indiquer le chemin vers le répertoire "`images`". Puis sélectionnez `Ajouter => Ajouter des fichiers` et naviguez jusqu'au fichier `icon_but.png`. Double-cliquez sur son nom pour l'ajouter dans le fichier de ressources (il apparaitra sur la liste en haut de la fenêtre). Maintenant ouvrez le fichier `mainwindow.cpp` et ajoutez dans la sous-routine `init()` la ligne suivante :

```
ui->icon_but->setIcon(QPixmap(":/images/icon_but.png"));
```

Ceci va mettre une icône sur le bouton `icon_but` via la fonction `setIcon`. , qui va tout d'abord convertir le fichier `*.png` en pixmap. Compilez le programme pour tester son fonctionnement.

Notez qu'il y a une autre possibilité d'ajouter une icône à un widget : on peut spécifier le fichier approprié dans l'éditeur de propriétés (voir aussi la Note au début du paragraphe III).

b) Nous allons maintenant inclure les ressources dans les feuilles de style. On va ajouter une image sur le fond du widget `textEdit`. Allez chercher une image appropriée en format *.png et sauvegardez le fichier dans le répertoire `images`. Supposons que son nom est `bg.png`. Ajoutez ce fichier aux ressources via `Ressource Editor`. Editez le fichier `mainwindow.cpp` en ajoutant à la fin de `init()` la ligne suivante :

```
ui->textEdit->setStyleSheet("background-image: url(:/images/bg.png)");
```

Compilez et exécutez le code. Vous devriez voir l'image contenu dans le fichier `bg.png` sur le fond de `textEdit`.

Note : Dans le cas d'une icône, la taille de l'image est ajustée à la taille du bouton. Donc, la taille finale est ca. 24 x 24 pixels (sans modification explicite), même si l'image original peut être beaucoup plus grand. Par contre, si on met l'image sur le fond d'un widget, la taille originale est préservée. Normalement il faudra donc ajuster l'image sélectionnée aux dimensions du widget utilisé, soit manuellement, soit en cochant l'option `scaledContents` dans l'éditeur de propriétés.

Si l'image est trop petite, ce serait mieux de l'étirer pour remplir l'espace du widget. C'est possible, mais dans ce cas il faudra définir l'image comme celle de la bordure et non du fond. Essayez ceci :

```
ui->textEdit->setStyleSheet("QTextEdit {border-width: 4px;"
                "border-image: url(:/images/bg.png) 4 4 4 4 stretch stretch}");
```

Pour comprendre les paramètres ci-dessus consultez la documentation (`Creating Scalable Styles`). Bref, on peut étirer l'image (`stretch`) ou le répéter (`tile`), ainsi que contrôler le comportement de coins.

III. Utilisation de la bibliothèque Qt

Dans cette partie nous allons aborder la création des interfaces graphiques en utilisant les éléments de la bibliothèque Qt. Ceci ne sera pas limité à l'emploi de nouveaux widgets, mais concernera surtout la fonctionnalité préprogrammée de la bibliothèque, ce qui nécessitera un certain minimum du codage.

Note : Parfois il y a certaines différences entre la version Qt(Windows) et Qt(Linux), probablement à cause de certaines bogues. Nous l'avons déjà vu sur l'exemple du cadre de QGroupBox. Il y a aussi quelques problèmes avec l'éditeur de propriétés sous Linux. Prenez l'exemple d'une `pixmap` ajoutée à une `QLabel`. Sous Linux, il faut parfois la mettre explicitement dans le code, via la fonction `setPixMap` (notez aussi que l'image devrait être en format PNG. Les autres, comme JPG, marchent sous Windows mais pas toujours sous Linux). Finalement, les liens externes configurés via l'option de l'éditeur de propriétés sous Linux ne marchent pas ; il faut utiliser la fonction `setOpenExternalLinks(true)`. Gardez ces remarques en tête en suivant les exemples ci-dessous.

1) Remarques générales

La bibliothèque Qt est très riche, tant en nombre d'objets graphiques (widgets) qu'elle fourni, qu'en fonctionnalités, contenant des milliers de fonctions et paramètres qui peuvent être librement personnalisés. Ceci peut vite conduire à un sérieux mal à la tête, d'où le système d'aide en ligne impressionnant. Cependant, il faut savoir que les développeurs de ce système ont fait des efforts considérables afin de s'assurer que l'ensemble des éléments de la bibliothèque soit intégré de façon cohérente avec la totalité des outils de développement habituels. En pratique, pour réaliser la fonctionnalité désirée d'une interface, on peut se servit majoritairement des outils et des éléments Qt, en ayant rarement recours à un langage spécifique (e.g. C ou C++). Ainsi, il faut savoir que le système Qt est basé sur le langage C++, organisé comme une bibliothèque orienté objet, contenant les définitions de variables en tant que structures (conteneurs). Ceci veut dire que, au-dehors de variables, les structures contiennent aussi les fonctions qui agissent sur ces variables.

Par exemple, prenons une chaîne de caractères (`string`). En langage C il existe plusieurs fonctions qui permettent de faire les opérations sur les variables de ce type, comme `strlen` (détermine la longueur de la chaîne), `strcpy` (copie une chaîne vers une autre), `strcat` (combine deux chaînes), `strcmp` (compare deux chaînes), etc. Dans le langage C++ on utilise plutôt les structures qui permettent de considérer les fonctions comme leurs membres. Pour récupérer la longueur d'une chaîne, en C il faut appeler une fonction avec l'argument qui pointe vers les caractères (`strlen(mystring)`), tandis qu'en C++ il suffit d'appeler la fonction `size` comme membre de la structure qui contient les caractères (`mystring.size()`). Dans le cas où `mystring` représente un pointeur, on peut accéder à son membre en utilisant l'opérateur flèche : `mystring->size()`. Il est important de savoir que dans le système Qt on a redéfini quasiment toutes les variables en classes de conteneurs et par conséquent, pour préserver la cohérence et faciliter la programmation, il est préférable de se servir de ces nouvelles définitions (e.g. `QString`, `QList`, `QStringList`, `QVector`, `QDate`, etc.).

En ce qui concerne les widgets, on doit prendre en charge beaucoup d'événements et prévoir ce qui va se passer quand on met le curseur de la souris là-dessus, quand on clique, quand on presse un bouton ou une touche du clavier, etc. Par conséquent, il existe dans la bibliothèque Qt des classes de fonctions qui répondent aux événements (e.g. `QKeyEvent` pour les actions du clavier, `QMouseEvent` pour la souris, `QActionEvent` quand on sélectionne un menu, `QTimer` pour les événements liés avec l'écoulement de temps, etc.). Consultez la documentions concernant les `Event Classes`.

Afin de pouvoir réaliser les opérations d'entré et de sortie, la bibliothèque Qt fourni les fonctions qui s'occupent de la lecture et écriture standard et/ou de fichiers (`QFile`, `QTextStream`), de la gestion de répertoires et de fichiers (`QDir`), du lancement de programmes externes (`QProcess`), d'utilisation de bibliothèques externes (`QLibrary`), etc. Voir les détails dans la documentation.

Les classes décrites ci-dessus sont regroupés dans les modules `QtCore` et `QtGui`. Il existe beaucoup d'autre classes de fonctions, regroupés dans d'autres modules de la bibliothèque Qt, qui s'occupent de la gestion de multimédia (`QtMultimedia`, `Phonon`), du développement réseau (`QtNetwork`), du graphisme 3D (`QtOpenGL`), de la gestion de bases de données (`QtSql`), du contenu Web (`QtWebKit`), du développement d'aide en ligne (`QtHelp`), et bien d'autres.

Nous allons explorer certaines de ces fonctionnalités sur les exemples détaillés ci-dessous.

2) Exemple de dialogues prédéfinis : `File Open`

Nous avons créé un menu `File` avec les options `Open` et `Save` (si vous avez déjà effacé le projet, recréez-le). Tout d'abord, nous allons apprendre à récupérer le nom du fichier sélectionné par l'utilisateur pour l'ouverture. Pour ce faire, il faut associer la sélection du menu `Open` avec la fonction appropriée. Allez dans l'éditeur des actions (en bas à gauche de la fenêtre du `qtcreator`) et faite un clic droit sur `actionOpen`. Sélectionnez `Go to Slot`, puis `triggered`. Le programme affiche la fonction `on_actionOpen_triggered()` du fichier `mainwindow.cpp`. Elle est aussi enregistrée en tant que `Private slot` dans `mainwindow.h` (vérifiez-le !). Pour voir si le déclenchement de cette fonction marche comme prévu, on peut la tester avec l'affichage de `QMessageBox`. Passons maintenant à la sélection d'un fichier. Mettez le code ci-dessous dans la fonction :

```
QString fileName = QFileDialog::getOpenFileName(this, "Open File",
                                                 "/home",
                                                 "Images (*.png *.xpm *.jpg)");
QMessageBox::about(this, "You selected file:", fileName);
```

La première ligne active le `File Open` dialogue via l'option `getOpenFileName`. Le widget parent est indiqué par le mot `this` (dans ce cas `MainWindow`). Le dialogue sera affiché au centre de ce widget (si vous tapez 0, ce sera au centre de l'écran). Le titre qui apparaît sur la fenêtre du dialogue est la chaîne "`Open File`", la fenêtre va montrer les fichiers qui se trouvent dans le répertoire "`/home`" et le filtre de fichiers les limite aux images du type PNG, XPM et JPG (le mot `Images` apparaîtra sur le bouton en bas de la fenêtre du dialogue).

Pour vérifier si le nom du fichier sélectionné a été correctement lu, on peut l'afficher dans la boîte `QMessageBox` (la deuxième commande ci-dessus). Toutefois, avant de l'utiliser dans le programme, il convient de s'assurer qu'il n'est pas vide :

```
if ( !fileName.isEmpty() )
{
      // do something
}
```

La fonction `isEmpty` vérifie si la chaîne `fileName` est vide. Si ce n'est pas le cas (`!` = négation), alors le programme va faire quelque chose.

Note : le dialogue est modal, c'est-à-dire il bloque l'accès à la fenêtre principale (celle avec le menu `File`). L'accès est rétabli quand on ferme le dialogue (en cliquant sur `OK` ou sur `Cancel`).

Maintenant abordons la tâche un peu plus compliquée. Supposons que nous souhaitons sélectionner plusieurs fichiers à la fois. Nous pouvons utiliser la même fonction, avec quelques petites modifications :

```
QStringList fileNames = QFileDialog::getOpenFileNames(this, "Open File",
                                                      "/home",
                                                      "Images (*.png *.xpm *.jpg)");
```

Comme vous voyez, le résultat n'est plus une chaîne (`QString`), mais une liste de chaînes (`QStringList`). Par conséquent, on spécifie l'option `getOpenFileNames` (en pluriel). Ceci active l'option de sélection multiple (on peut la réaliser en cliquant sur les noms de fichiers et en appuyant en même temps sur la touche `Ctrl`). Les autres options ont la même signification.

Pour afficher les résultats (plusieurs noms de fichiers sélectionnés), on va aussi utiliser un `QMessageBox`. Puisque cette fonction a besoin d'une chaîne comme paramètre d'entrée, il faudra convertir la liste de chaînes en une seule chaîne, dans laquelle les noms de fichiers seront séparés par "`\n`" (équivalent de la touche "`Entrée`" du clavier). Le code ci-dessous accompli cette tâche :

```
QString s;
for (int i = 0; i < fileNames.size(); ++i)
{
    s.append(fileNames.at(i));
    s.append("\n");
}
QMessageBox::about(this, "You selected files:", s);
```

Une simple boucle permet d'ajouter (`append`) les noms de fichiers un par un (indice `at(i)`) et d'insérer le séparateur "`\n`". Finalement, on affiche le résultat. Vérifiez comment ça marche.

Note : pour utiliser les filtres de fichiers multiples, séparez-les par "`;;`". Par exemple, pour pouvoir choisir entre les images, les fichiers texte, et les fichiers Word, utilisez le filtre suivant :

```
"Images (*.png *.xpm *.jpg);;Text files (*.txt);;DOC files (*.doc)"
```

Ceci affichera le menu déroulant sur le bouton au-dessous de la liste de fichiers.

3) Exemple de dialogues prédéfinis : `File Save`

La procédure d'utilisation du dialogue `QFileDialog` est la même que dans le cas précédent, sauf que le membre de la classe `getOpenFileName` est remplacé par `getSaveFileName`. Les autres paramètres ont la même signification. Voici l'exemple du code :

```
QString fileName = QFileDialog::getSaveFileName (this, "Export PDF file",
                                                  "/home",
                                                  "Images (*.png *.xpm *.jpg)");
```

Afin de vérifier si le nom est valide (si l'utilisateur annule le dialogue, le nom retourné est vide) on appelle la fonction `isEmpty()`. D'autre côté, on tape souvent le nom sans extension, mais en cas d'oubli, elle peut être ajoutée dans le code :

```
if ( !fileName.isEmpty() )
{
    if (QFileInfo(fileName).suffix().isEmpty()) fileName.append(".pdf");
}
```

Comme avant, affichez le résultat pour s'assurer que tout se passe bien.

Note : Contrairement au cas précédent, il n'est pas nécessaire que le fichier sélectionné existe (on peut souhaiter de sauvegarder les données dans un nouveau fichier).

4) Exemple de dialogues prédéfinis : messages d'erreur

En principe, pour afficher les messages d'erreur on peut se servir de `QMessageBox` avec l'option `warning` ou `critical`. Cependant, il existe une classe `QErrorMessage`, qui permet d'afficher un message avec une option à cocher (`Check Box`). Elle permet d'éviter l'affichage d'une série de messages identiques. Un exemple d'utilisation de cette classe est montré ci-dessous :

```
QErrorMessage errorMessage;
errorMessage.showMessage("Testing Error message");
 errorMessage.exec();
```

5) Utilisation des onglets (Tabs)

Le `Tab Widget` permet de créer un panneau avec les onglets. Nous allons voir comment les créer, modifier et gérer. En même temps nous allons continuer l'exploration de dialogues prédéfinis, sur les exemples concernant les polices (`QFontDialog`), les couleurs (`QColorDialog`) et les imprimantes (`QPrintDialog`). Nous allons créer un dessin minimal pour faire une sélection et afficher les résultats. Commençons par l'ajout d'un `Tab Widget` sur l'interface graphique actuel. Donnez-lui le nom `sel_tabs`. Par défaut, on voit deux onglets. Cliquez sur le deuxième pour le sélectionner, puis faites un clic droit sur l'espace vide de cet onglet et choisissez l'option `Insert page => After current`. Maintenant on a trois onglets. Cliquez sur le premier, et dans l'éditeur de propriétés trouvez `currentTabText` et tapez "`Font`". Puis, changez le nom du deuxième onglet à "`Colors`" et du troisième à "`Printer`".

Dans l'espace du premier de trois onglets (`Font`) introduisez un widget du type `QLineEdit` et un du type `Push Button`. Nommez le premier `font_sel` et le deuxième `font_but`. Remplacez le texte du bouton par "Select font". Pour l'onglet `Colors`, faites pareil (deux widgets de mêmes types que ceux de l'onglet `Font`) et nommez-les `color_sel` et `color_but`. Pour le widget `Line Edit` sélectionnez l'alignement du texte au centre. Mettez le texte du bouton à "Pick colors". Sur le dernier onglet (`Printer`), placez un `textEdit` (l'impression que nous allons essayer ne marchera pas sur un `QLineEdit`) et un `Push Button`. Nommez les deux widgets `print_sel` et `print_but`, avec le texte du bouton "Choose printer". Si le texte dépasse la taille d'un de boutons, redimensionnez le widget. Notre objectif sera d'afficher un sélecteur approprié après un clic sur un bouton et d'afficher le résultat de la sélection dans le champ de `Line Edit` (ou `textEdit`, pour l'onglet `Printer`) correspondant.

a) Sélection de police

Sélectionnez le premier onglet (`Font`) et faites un clic droit sur le bouton. Sélectionnez l'option `Go To Slot`, puis `clicked()` et `OK`. Ceci crée une fonction `on_font_but_clicked` dans le fichier

`mainwindow.cpp`. Comme avant, notez que `qtcreator` a ajouté un `private slot` de ce nom dans le fichier `mainwindow.h`. Dans le corps de cette fonction ajoutez l'appel au dialogue de polices :

```
bool ok;
QFont font = QFontDialog::getFont(&ok, QFont("Times", 12), this);
```

Cet appel propose une police par défaut (ici `Times, 12 pt`) et attend le résultat dans l'objet `font` du type `QFont`. La variable booléenne `ok` renseigne si l'utilisateur a cliqué sur le bouton OK (`ok = TRUE`) ou sur Cancel (`ok = FALSE`). En fonction de la valeur `ok` l'objet `font` contient soit la police sélectionnée par l'utilisateur, soit la police proposée comme police par défaut (`Times, 12`). Normalement, on effectuerait un branchement dans le programme comme dans le code suivant :

```
if (ok) {
    // font is set to the font the user selected
} else {
    // the user cancelled the dialog; font is set to the initial
    // value, in this case Times, 12.
}
```

Pour visualiser la sélection, il suffit d'extraire la chaîne descriptive concernant la police sélectionnée de l'objet `font`. Puis, de la mettre dans le champ de `Line Edit` s'appellant `font_sel` :

```
QString str = font.toString();
ui->font_sel->setText(str);
```

En pratique, on peut combiner les deux instructions en une seule, ainsi évitant la déclaration explicite de la chaîne `str` :

```
ui->font_sel->setText(font.toString());
```

b) Sélection de couleurs

Maintenant on passe au deuxième onglet (`Colors`). Comme avant, créez le slot qui sera exécuté après un clic sur le bouton (`on_color_but_clicked`) et mettez-y le code suivant :

```
QColor color;
color = QColorDialog::getColor(Qt::green, this);
```

Le premier argument spécifie la couleur initiale qui sera affiché dans la boîte de dialogue. Quand l'utilisateur ferme le dialogue, le résultat est sauvegardé dans l'objet `color` du type `QColor`. Il ne reste que vérifier s'il est valide, et si oui, afficher le nom de la couleur dans le champ de LineEdit correspondant :

```
if (color.isValid()) {
      ui->color_sel->setText(color.name());
}
```

Le texte devrait apparaître au centre du widget, parce que nous avons choisi cet alignement dans l'éditeur de propriétés. Pour faire plus joli, nous allons mettre la couleur sélectionnée au fond du widget `color_sel`. Ajoutez au code ci-dessus les lignes suivantes :

```
QString s = QString("QLineEdit#color_sel {background-color: %1}")"
                    ".arg(color.name());
ui->color_sel->setStyleSheet(s);
```

Ici, on se sert encore une fois de `Style Sheets` pour spécifier la couleur du fond. Dans la définition de la chaîne on restreint la portée de cette feuille de style au widget actuel, en ajoutant au mot-clé `QLineEdit` l'identificateur du widget de ce type : `QLineEdit#color_sel` (sans cet identificateur le style s'appliquerait à tous les widgets du type `QLineEdit`). Les premières deux lignes montrent l'utilisation de la commande `QString.arg`, qui fournit la fonctionnalité de la commande `sprintf` en langage `C`. Elle permet donc d'écrire dans une chaîne (`s`) une liste des arguments de façon formatée. Le symbole `%1` signifie le premier argument sur la liste, qui est ici `color.name()`. Pour les détails, consultez la documentation.

c) Sélection d'imprimante

Pour choisir une imprimante de sortie de données, on peut se servir du dialogue `QPrintDialog`. Voilà l'exemple de son utilisation :

```
QPrinter printer;
QPrintDialog dlg(&printer, this);
if( dlg.exec() == QDialog::Accepted )
{
    QString s = printer.printerName();
    QMessageBox::information(this, "Selected printer", s);

}
```

Pour commencer, on déclare `printer`, l'objet du type `QPrinter`, qui représente une classe très riche en options (voir la documentation pour une description détaillée). Puis, on crée l'objet `dlg` du type `QPrintDialog`, associé avec le `printer`. L'exécution du dialogue (fonction `exec()`) permet de définir toutes les options du `printer` (la taille de page, son orientation, résolution...), si l'utilisateur a fait un choix valide (`QDialog::Accepted`). Nous pouvons vérifier la sélection d'utilisateur en affichant le nom de l'imprimante (`printerName`) dans `QMessageBox`.

Ceci dit, il serait mieux d'imprimer effectivement quelque chose. Il serait difficile de parler de toutes les options disponibles, surtout parce qu'elles dépendent souvent du type de sortie, donc nous allons nous limiter à la chose basique : l'impression du contenu de la fenêtre courante (c'est-à-dire celle sur laquelle nous avons poussé le bouton `Print`). Remplacez l'affichage de `QMessageBox` par le code suivant :

```
QPainter painter(&printer);
this->render(&painter);
painter.end();
```

La classe `QPainter` est trop riche pour en discuter les détails ici (regarder la documentation). Elle sert à créer les graphismes, même très complexes. Nous l'utilisons ici pour transférer le contenu de la fenêtre actuelle vers l'imprimante sélectionnée grâce à la fonction `render`. Pour des raisons pratiques il serait peut-être raisonnable dans cet exercice de sélectionner comme imprimante le créateur de fichiers PDF, s'il est installé sur votre machine.

En fait, on peut définir le type de sortie sans passer par le dialogue du choix d'imprimante. Voici l'exemple du code qui configure la sortie en format PDF.

```
QPrinter printer(QPrinter::HighResolution);
printer.setOutputFormat(QPrinter::PdfFormat);
printer.setOutputFileName(fileName);
printer.setPaperSize(QPrinter::A4);
printer.setOrientation(QPrinter::Portrait);
printer.setPageMargins(15, 10, 15, 10, QPrinter::Millimeter);
```

Ci-dessus, les chiffres signifient les marges (gauche, haut, droit, bas) en mm. Il ne reste que l'impression :

```
ui->print_sel->print(&printer);
```

Notez bien que, en fonction du type de widget, au lieu d'imprimer directement son contenu comme dans l'exemple ci-dessus, il faut parfois passer par l'impression du document qui en fait partie, ce qui montre la ligne suivante :

```
ui->print_sel->document()->print(&printer);
```

Pour un objet du type `textEdit`, ces deux options donnent le même résultat.

6) Menus contextuels

Menus contextuels sont ceux qui apparaissent quand on clique avec le bouton droit de la souris sur un objet de l'interface graphique. On peut ainsi présenter une liste des options à choix dont une sera exécutée après la sélection via un clic du bouton gauche. Ces menus sont contextuels parce qu'ils peuvent être définis indépendamment pour chacun des objets de l'interface (le menu de la fenêtre principale – voir II/6 – est unique pour toute l'interface).

Nous allons suivre les étapes de la création d'un menu contextuel sur l'exemple d'un widget du type `textEdit`, mais la même procédure est valide pour n'importe quel autre objet. Tout d'abord, ajoutons un widget à l'interface. Supposons que son nom est `textEdit`. Dans l'éditeur de propriétés, trouvez la propriété `ContextMenuPolicy` et sélectionnez l'option `CustomContextMenu`. Faites un clic droit sur le widget en question et dans la liste qui apparait (notez bien que cette liste représente justement un menu contextuel !) sélectionnez `Go To Slot`. Dans le dialogue qui s'ouvre, sélectionnez `CustomContextMenuRequested(QPoint)`, puis `OK`. Le programme vous affiche la fonction nouvellement créée `on_textEdit_customContextMenuRequested` dans le fichier

`mainwindow.cpp` qui s'exécutera chaque fois quand on demandera le menu contextuel de ce widget. Apercevez aussi que cette fonction a été ajoutée dans `mainwindow.h` en tant que `Private slot`.

La compilation va réussir, mais rien ne se passera sans définition du contenu de notre menu contextuel. Pour ce faire, il suffit de créer une liste de chaînes (`QStringList`), où chaque ligne correspond à une option du menu. Par exemple, on peut mettre à l'intérieur de notre nouveau slot le code suivant :

```
QStringList cont_menu;
cont_menu << "1st line";
cont_menu << "2nd line";
cont_menu << "3rd line";
```

Ici, on a ajouté 3 lignes de texte dans la liste de chaines `cont_menu`. L'étape suivante est d'enregistrer la liste `cont_menu` en tant que contenu d'un menu des actions :

```
QMenu my_menu;
for (int i = 0; i < cont_menu.size(); ++i)
      my_menu.addAction(cont_menu.at(i));
```

Ci-dessus on déclare un objet du type `QMenu` et on y ajoute dans une boucle, ligne par ligne, le contenu de `cont_menu`, à partir de la première ligne jusqu'à la dernière, dont l'indice est donné par la fonction `cont_menu.size()`. La fonction `my_menu.addAction` signale au système que le texte indiqué par l'indice `cont_menu.at(i)` doit être traité comme une option sélectable dans le menu contextuel.

Quand le code de la fonction `on_textEdit_customContextMenuRequested` est exécuté, ceci signifie que l'utilisateur a déjà fait un clic droit sur le widget. La question est où exactement ? Normalement, le menu apparaît sous le curseur de la souris, donc il faut connaitre sa position. En fait, la position est passée à notre fonction sur la liste de paramètres de l'appel en tant qu'un jeu de coordonnées du point où se trouve le curseur (`QPoint &pos`). Mais ces coordonnées sont relatives par rapport au widget, et nous souhaitons afficher le menu dans les coordonnées relatives à l'écran. Une conversion est nécessaire, d'où le code suivant :

```
QPoint pt = this->textEdit->mapToGlobal(pos);
QAction* sel_item = my_menu.exec(pt);
```

Dans la première ligne nous créons un nouveau point `pt`, obtenu du point `pos` via la fonction de conversion `mapToGlobal`. Puis, on exécute `my_menu` (`my_menu.exec`) en l'affichant à la position `pt`. Cette fonction va se terminer après un clic d'utilisateur, en indiquant l'action sélectionnée. Celle-ci sera lue dans l'objet `sel_item`, déclaré comme `QAction`. Il ne reste qu'à décoder quelle option a été choisie et réagir de façon appropriée. Tout d'abord, vérifions si un choix a été fait (l'utilisateur a pu cliquer ailleurs, ce qui équivaut l'annulation du menu) :

```
if (sel_item)
{
      // something was chosen, do stuff
}
else
{
      // nothing was chosen
}
```

Pour vérifier ce que l'utilisateur a choisi; nous allons simplement afficher l'option sélectionnée dans un `QMessageBox`. Il faut extraire le texte de l'option sélectionnée de l'objet `sel_item`, comme suit :

```
QString s;

if (sel_item)
{
      s = sel_item->text();
      QMessageBox::information(this, "You selected:", s);
}
else
{
      // nothing was chosen
}
```

Tout est prêt à recevoir les souhaits de l'utilisateur. Il ne reste qu'à coder ce qu'on souhaite faire en fonction de ses choix. Dans le cas typique, on va remplacer la ligne `QMessageBox` ci-dessus par une série des instructions conditionnelles afin de décider sur l'action à mener :

```
if ( s == cont_menu.at(0))
{
      // do something if 1st line selected
}

if ( s == cont_menu.at(1))
{
      // do something if 2nd line selected
}
…
```

Note : Dans cet exemple, le contenu du menu a été créé à l'intérieur du slot, de façon dynamique. On pourrait bien imaginer que ce contenu va évoluer en fonction d'autres choix d'utilisateur. Si vous souhaitez un menu statique, on peut le créer en-dehors du slot et simplement lire son contenu au moment de la création du menu contextuel.

7) Travail avec plusieurs fenêtres

Très souvent, quand on clique sur un menu ou sur un objet, ceci entraine l'apparition d'une nouvelle fenêtre. L'exemple le plus simple est celui du choix de l'option "`Help => About`" dans le menu principal de l'interface graphique. D'habitude on voit apparaitre un dialogue explicatif sur l'application, les auteurs, etc. Nous allons apprendre comment ajouter une nouvelle fenêtre à notre interface et comment assurer la communication entre elles.

- Dans le menu `File => New` choisissez l'option `Qt` sous `Files and classes`, puis à droite `Qt Designer Form Class`. Dans le dialogue suivant, sélectionnez `Dialog with Buttons Bottom`. Donnez un nom au nouveau dialogue, par exemple `About`. Ajoutez le nouveau dialogue au projet actuel et terminez. Vous devriez voir plusieurs nouveaux fichiers dans le listing à gauche : `about.h`, `about.cpp` et `about.ui`.

- Ouvrez la nouvelle fenêtre en double-cliquant sur `about.ui`. Ajoutez-y deux widgets du type `QLabel`. Nous allons utiliser le premier pour afficher un logo. Nommez-le `about_img`. Le deuxième sera utilisé pour afficher quelques lignes de texte. Nommez-le `about_info`.

- Créez un fichier de ressources et ajoutez-le au projet actuel. Puis, cherchez une image PNG qui jouera le rôle du logo et ajoutez-la au fichier de ressources. Cliquez sur `about_img` pour le sélectionner, puis cherchez dans l'éditeur de propriétés l'option `pixmap` et choisissez votre image (**attention : sélectionnez l'image des ressources, et non celle du fichier externe**). Vous pouvez aussi cochez l'option `scaledContents`.

- Il est temps de lier la nouvelle fenêtre `About` à l'interface existante. Le problème c'est que chacune de classes (ici `MainWindow` et `About`) est séparée des autres et ses membres ne sont pas accessibles de l'extérieur. Comment donc afficher la fenêtre `About` en cliquant sur l'action `About` du menu `Help`, qui appartient à la fenêtre `MainWindow` ?

  Nous allons utiliser une astuce, basée sur la création de pointeurs vers les classes différentes, qui, eux, seront déclarés comme variables globales. Par conséquent, chaque fenêtre pourra accéder à tous les membres d'autres fenêtres via leurs adresses, stockées dans les pointeurs. Voici comment s'y prendre.

  En premier temps, nous allons modifier le fichier `main.cpp`. Pour assurer l'accès à la définition de notre nouvelle fenêtre, il faudra ajouter une ligne en haut du fichier :

  ```
  #include "about.h"
  ```

  Au-dessous de ces lignes (`#include`) mais avant la ligne "`int main`" nous allons ajouter la déclaration d'un pointeur, qui sera initialisé à la valeur `NULL`. Notez bien que, puisque cette déclaration est faite au-dehors du code du programme `main`, ce pointeur aura la portée globale :

  ```
  void *ptr2ab = NULL;
  ```

  Dans le code du programme `main`, avant `w.show`, nous allons ajouter les lignes suivantes :

  ```
  About ab;
  ptr2ab = &ab;
  ab.ab_init();
  ```

  Ceci veut dire que nous créons le dialogue `ab` de la classe `About` et stockons son adresse (le symbole `&`) dans le pointeur `ptr2ab`. La dernière ligne représente un appel à une fonction d'initialisation de cette nouvelle fenêtre (`ab_init`), dans laquelle nous allons mettre tout ce qu'il faut afficher, mais qui reste encore à déclarer et à écrire.

- L'étape suivante, c'est la modification du fichier `about.h`. Elle est minuscule, il s'agit simplement d'ajouter la fonction `ab_init` en tant que `public slot`. Ajoutez deux lignes suivantes avant la fin de déclarations :

  ```
  public slots:
      void ab_init();
  ```

- Nous allons enfin créer la fonction `ab_init()`. Ajoutez le template vide de cette fonction à la fin du fichier `about.cpp` :

  ```
  void About::ab_init()
  {
  }
  ```

- Notez bien que jusqu'à maintenant nous avons travaillé sur la préparation du dialogue `About` pour l'interaction avec la fenêtre principale. L'étape finale, c'est de rendre possible l'affichage de ce dialogue en cliquant sur `Help => About`. Pour ce faire, double-cliquez sur `mainwindow.ui` et allez dans l'éditeur des actions. Faites un clic-droit sur `actionAbout` et sélectionnez `Go To Slot`, puis `triggered()`. Vous verrez s'afficher le template vide de la fonction `on_actionAbout_triggered` à la fin du fichier `mainwindow.cpp`. Puisque nous souhaitons travailler avec le dialogue `About`, il faudra procéder de façon identique comme dans le cas du fichier `main.cpp`, c'est-à-dire d'ajouter la ligne

  ```
  #include "about.h"
  ```

  en haut du fichier, suivie de la déclaration du pointeur `ptr2ab`. Puisque le pointeur a été déjà déclaré dans `main.cpp`, ici on va seulement indiquer qu'il est externe à ce module. Le programme va aller chercher sa définition ailleurs :

  ```
  extern void *ptr2ab;
  ```

  Revenons à la fonction `on_actionAbout_triggered`. Pour afficher le dialogue `About`, nous allons récupérer son adresse du pointeur `ptr2ab`, puis afficher la nouvelle fenêtre :

  ```
  About* ab = (About*) ptr2ab;
  ab->show();
  ```

  La première ligne déclare le dialogue `ab` du type `About`, et spécifie qu'il faut allez chercher sa définition à l'adresse indiqué par le pointeur `ptr2ab`. Le programme devrait se compiler sans problème. Ceci dit, l'objet `about_info` sera vide, parce que pour l'instant nous n'avons rien fait pour y ajouter un texte. Nous allons remédier à ce problème. Ajoutons du code à la fonction `ab_init()` dans le fichier `about.cpp`. Commençons par un exemple basique :

  ```
  ui->about_info->setText("This is just a test string.");
  ```

  Recompilez le programme et lancez-le. Affichez le dialogue `About` et admirez votre création. Ceci dit, l'information dans `about_info` n'est pas très riche. Rassurez-vous, on peut faire beaucoup mieux.

- Pour compliquer un peu les choses, nous allons modifier le texte de l'étiquette `about_info` de façon à introduire les liens. D'habitude, quand on place le curseur de la souris sur un lien, le curseur change la forme, et quand on clique là-dessus, ceci lance une action qui dépend de la nature de ce lien. Nous allons explorer deux cas : ouverture d'une page Web et l'envoie d'un message email via un programme externe.

  Souvenez-vous que dans la bibliothèque Qt on est autorisé à utiliser le langage HTML dans les textes. Nous allons en profiter. Examinez le code ci-dessous :

  ```
  QString info("<html><b><p style='font-family: arial; font-size:9pt;"
               "font-style: normal'>"
               "This is a sample text to be shown in the About window.<br>"
               "Send suggestions and bug reports to:<br>"
               "<br>"
               "<a href='mailto:cgeorge@ipbs.fr?subject=Qt course feedback&body='>"
               "cgeorge@ipbs.fr</a><br>"
               "<br>"
               " or visit the "
               "<a href='http://www.ipbs.fr'>IPBS</a> Web site."
               "</p></b></html>");
  ui->about_info->setText(info);
  ```

Dans cet exemple au début on définit le type et la taille de la police, et après on insère le texte. Pour définir un lien, on s'appuie sur la syntaxe HTML (voir la documentation ailleurs, elle est indépendante du système Qt). Notez bien que, dans le cas d'un email, il est possible de spécifier le destinataire (`mailto`), le sujet (`subject`) ainsi que le corps (`body`, ici vide) du message à envoyer. Dans les deux cas le texte représentant la définition du lien reste invisible ; ce qui s'affiche c'est le texte qui suit le tag HTML correspondant, et il est souligné.

Le dernier point important est de s'assurer que les liens externes seront ouverts par le programme. Cherchez l'option `openExternalLinks` dans l'éditeur de propriétés et cochez la case.

Note : Après un clic sur un lien, votre programme essaie de lancer un navigateur (ou un programme de courrier électronique) qui est défini comme programme par défaut du votre système. Si rien ne se passe, il est probable qu'il y a un problème avec la configuration du système. Par exemple, sous Linux le programme `evolution` est souvent défini comme programme Email par défaut. Mais si vous utilisez Thunderbird, il faut alors le signaler au système (e.g. tapez la commande "`thunderbird -setDefaultMail`"). En général, sous Fedora 19 vous pouvez le faire en allant dans le menu `Applications => Settings => Details => Default applications`.

8) Accès aux fichiers et aux répertoires

Les exemples ci-dessous permettent de comprendre comment afficher le contenu d'un répertoire, comment ouvrir l'accès à un fichier, lire son contenu, le changer et finalement fermer l'accès.

a) Affichage du contenu d'un répertoire

Afin d'aborder ce sujet, placez sur la fenêtre graphique deux objets du type `QListWidget`. Nommez le premier `dir_list` et le deuxième `file_list`. Notre objectif sera d'afficher la liste de fichiers contenus dans un répertoire quand on le sélectionne. Créez une fonction d'initialisation pour la fenêtre principale de même façon qu'auparavant :
- ajoutez la ligne "`w.init()`" avant "`w.show()`" dans `main.cpp`
- ajoutez la ligne "`void init()`" dans `public slots` du fichier `mainwindow.h`
- créez la fonction `init()` vide dans le fichier `mainwindow.cpp`

Maintenant vous pouvez ajouter le code suivant dans le corps de la fonction d'initialisation :

```
ui->dir_list->clear();
QDir path(".");
ui->dir_list->addItem(path.absolutePath());
```

La première instruction efface le contenu de l'objet `dir_list`, la deuxième déclare un chemin (`path` du type `QDir`) qui correspond au répertoire courant (`"."`), la troisième ajoute ce chemin au contenu de `dir_list`. Notez que vous pouvez choisir entre deux options : le chemin absolu (`absolutePath()`) ou relatif (`path()`). Si vous voulez avoir un choix parmi plusieurs répertoires, il faudra les ajouter à la liste `dir_list` (expérimentez avec le code).

Note : On peut choisir entre la sélection d'un seul `item` sur la liste, et la sélection multiple. Il suffit de définir la propriété `selectionMode` dans l'éditeur de propriétés, ou le faire dans le code en utilisant la fonction : `setSelectionMode()` avec, comme argument, le mot clé `QAbstractItemView::SingleSelection` ou `QAbstractItemView::MultiSelection`.

L'étape suivante est de réagir à une sélection dans la liste `dir_list`. Faites un clic droit sur `dir_list` et sélectionnez `Go To Slot`, puis `itemClicked(QListWidgetItem *)`. Ceci activera la fonction `on_dir_list_itemClicked` en lui passant sur la liste d'arguments l'adresse (pointeur `*`) du `item` sélectionné. Notez bien qu'un `private slot` du même nom est apparu dans le fichier `mainwindow.h`. Cependant, l'`item` du type `QListWidgetItem` qui apparaît sur la liste des arguments risque de ne pas être reconnu, et donc il faut ajouter la ligne "`#include <QtGui>`" en haut de ce fichier. Dans la fonction correspondante créée dans le fichier `mainwindow.cpp` ajoutez ceci :

```
QString sel_path = item->text();
QDir path(sel_path);
QStringList list = path.entryList(QDir::Dirs | QDir::NoDotAndDotDot);

ui->file_list->clear();
ui->file_list->addItems(list);
ui->file_list->sortItems();
```

Ici, nous créons une chaîne `sel_path` qui correspond au nom du répertoire sélectionné par l'utilisateur, puis un répertoire `path` qui pointe vers cette sélection, et enfin nous ajoutons à la liste `list` les noms de fichiers qui se trouvent dans ce répertoire (via la fonction `entryList()`). Notez qu'on peut éviter l'affichage du répertoire courant ("`.`") ainsi que son parent ("`..`") en spécifiant le mot-clé `NoDotAndDotDot`. Pour finir, nous effaçons le contenu du widget `file_list` et ajoutons à son contenu la liste `list` qui vient d'être créée. Pour s'y mieux retrouver, on peut trier le contenu avec la fonction `sortItems()`.

b) Accès aux fichiers

Nous allons apprendre comment accéder aux fichiers en utilisant la fonctionnalité de la bibliothèque Qt sur un exemple pratique, celui d l'enregistrement et de la récupération de la position et de la taille actuelles de la fenêtre principale. Tout d'abord, ajoutez à votre interface graphique un champ du type `TextEdit` et deux boutons, avec les textes "`Save defaults`" et "`Read defaults`". Nommez-les `save_but` et `read_but`, respectivement.

Ensuite, créez les slots pour les deux boutons, qui s'activent quand on fait un clic là-dessus (vous devriez être déjà capables de le faire sans explications détaillées). Editez le fichier `mainwindow.cpp` et ajoutez les lignes suivantes avant la fonction `on_save_but_clicked()` :

```
char const ini_file[]="app.ini";
int scr_width = 0;
int scr_height = 0;
int app_width = 0;
int app_height = 0;
int app_x = 0;
int app_y = 0;
```

En bref, on initialise certains variables, qui vont contenir le nom du fichier avec les paramètres à sauvegarder, la largeur et la hauteur de l'écran, la largeur et la hauteur de la fenêtre de notre application, ainsi que la position X et Y de cette dernière (en comptant à partir du coin en haut à gauche de l'écran). Ces variables auront la portée globale dans le programme.

Dans la fonction `on_save_but_clicked()`, ajouter ceci :

```
QFile file(ini_file);
if (!file.open(QIODevice::WriteOnly | QIODevice::Text))
{
    QMessageBox::warning(this,"Error in saving defaults",
                         "Cannot open init file for writing!");
}
else
{
    app_width = width();
    app_height = height();
    app_x = geometry().x();
    app_y = geometry().y();
    QTextStream out(&file);
    out << app_width << endl;
    out << app_height << endl;
    out << app_x << endl;
    out << app_y << endl;
    file.close();
}
```

Le code est facile à comprendre : nous déclarons une variable du type `QFile` de nom spécifié auparavant. Ce fichier contient la taille et la position de notre interface graphique. Puis nous ouvrons ce fichier en mode `WriteOnly` et `Text`, et si ceci échoue, nous affichons un message d'erreur. Si tout va bien, nous initialisons les cellules de mémoire `app_width`, `app_height`, `app_x` et `app_y` avec l'information obtenue des fonctions appropriées de la bibliothèque Qt. Finalement, nous ouvrons un flux de sortie `out` (`QTextStream`) associé avec le fichier `file` et nous y écrivons sur quatre lignes consécutives l'information à sauvegarder (`endl` = end of line, c-à-d `Entrée`). A la fin nous fermons l'accès à ce fichier *via* la fonction `close()`.

L'information sauvegardée après un clic sur "`Save defaults`" sera lue quand on clique sur "`Read defaults`". Dans la fonction `on_read_but_clicked`, ajouter le code ci-dessous :

```
QFile file(ini_file);

if (!file.open(QIODevice::ReadOnly | QIODevice::Text))
{
    // set default position and size of the app's window to current values
    scr_width = QApplication::desktop()->width();
    scr_height = QApplication::desktop()->height();
    app_width = scr_width * 0.5;
    app_height = scr_height * 0.5;
    app_x = (scr_width - app_width) / 2;
    app_y = (scr_height - app_height) / 2;
    setGeometry(app_x,app_y,app_width,app_height);
}
else
{
    QTextStream in(&file);

    QString line;
```

```
    line = in.readLine();
    ui->textEdit->setText(line);
    app_width = line.toInt();

    line = in.readLine();
    ui->textEdit->append(line);
    app_height = line.toInt();

    line = in.readLine();
    ui->textEdit->append(line);
    app_x = line.toInt();

    line = in.readLine();
    ui->textEdit->append(line);
    app_y = line.toInt();

    setGeometry(app_x, app_y, app_width, app_height);

    file.close();
}
```

Encore une fois, c'est facile de comprendre ce qui se passe : nous ouvrons le fichier `ini_file` en mode `ReadOnly` et `Text`, et s'il n'existe pas (par exemple, parce que c'est la première utilisation, ou parce que quelqu'un l'a effacé), alors on exécute quelques instructions qui vont créer les valeurs par défaut. Concrètement, on fait appel aux fonctions `desktop()->width()` et `desktop->height()` pour récupérer la taille de l'écran. Puis, nous calculons la taille de notre application (qui occupera la moitié d'écran) et sa position (au milieu d'écran). Nous appelons la fonction `setGeometry` pour fixer ces paramètres avant l'affichage de l'interface.

Si le fichier a été ouvert correctement, on ouvre un `TextStream in` pour la lecture et on lit ligne par ligne, les quatre valeurs sauvegardées. Notez que, à chaque lecture, la valeur lue est copiée dans le `TextEdit`. Les valeurs numériques sont obtenues grâce à la conversion du texte vers les valeurs entières via la fonction `toInt()`. A la fin, on appelle la fonction `setGeometry` qui utilise les valeurs de quatre variables pour remettre la fenêtre graphique là où elle était quand on a cliqué sur "`Save defaults`".

Compilez le programme et testez-le : placez la fenêtre où vous voulez et redimensionnez-là comme bon vous il semble. Cliquez sur "Save defaults". Puis, déplacez la fenêtre ailleurs et changez sa taille. Cliquez sur "Read defaults". La fenêtre devrait revenir à la position et taille initiales. Notez que les paramètres actuels de la fenêtre sont affichés dans le champ de l'objet `textEdit`.

Exercice : En combinant vos connaissances en matière de la gestion de répertoires et de fichiers, créez une application qui affiche le contenu d'un répertoire, et - quand on clique sur le nom d'un fichier - affiche le contenu de ce fichier dans un champ `TextEdit` (admettons qu'il s'agit d'un fichier texte). Evidemment, vous pouvez vous inspirer du contenu de ce chapitre.

9) Miscellanées

La bibliothèque Qt étant très riche, il n'est pas possible d'en explorer toute la richesse en temps limité à quelques séances. Vous devriez consulter la documentation et aussi visiter les forums de discussion pour apprendre les aspects divers de son utilisation (voir les références à la fin de ce document). Ci-dessous vous trouverez quelques exemples d'utilisation des fonctions un peu plus "exotiques" mais néanmoins très utiles dans la conception d'une interface graphique.

a) changer le titre d'une fenêtre

Si vous voulez le faire une seule fois au début de l'exécution du programme, alors il suffit de chercher `windowTitle` dans l'éditeur de propriétés et taper le texte souhaité. Mais on peut changer le titre de façon dynamique, pendant l'exécution du programme, en appelant la fonction `setWindowTitle` :

```
this->setWindowTitle("New Title");
```

b) obtenir la taille d'une fenêtre et sa position

Souvent on souhaite sauvegarder la position et la taille de la fenêtre graphique affichée sur l'écran avant de la fermer, pour pouvoir la récupérer au moment du lancement suivant. Ceci peut se faire en utilisant les fonctions `pos` (position) et `size` (taille) de façon suivante (en supposant que la fenêtre s'appelle `mywin`) :

```
QPoint p = ui->mywin->pos();
QSize s = ui-> mywin ->size();
```

Les coordonnées stockées dans les objets `p` et `s` peuvent être sauvegardées dans un fichier externe. Quand l'application est lancée de nouveau il suffit de lire le contenu de ce fichier pour récupérer ces mêmes coordonnées. Si on va les lire dans les variables de même nom (`p` et `s`), on pourra mettre le code suivant juste avant le premier affichage de la fenêtre :

```
ui-> mywin ->resize(s);
ui-> mywin ->move(p);
 ui-> mywin ->show();
```

Note : Il existe une option plus "professionnelle" pour gérer ce problème, la classe `QSettings`. Ses membres permettent de préserver et de récupérer les paramètres d'un programme. Sur la plateforme Windows, l'information est sauvegardée dans le fichier de la base de registre, tandis que sous Linux elle est sauvegardée dans un fichier texte. Pour en savoir plus, consultez la documentation.

c) écrire dans la barre d'état

La ligne en bas de la fenêtre porte le nom de la barre d'état (*status bar*) et sert à afficher l'état du programme et/ou les messages importants. On peut l'accéder via la fonction `statusBar` de façon suivante :

```
QString s = tr("Right-click to display context menus");
QFont font("Helvetica",0,75,false);  // 0=system size, 75=bold, italic=false
this->statusBar()->setFont(font);
this->statusBar()->showMessage(s);
```

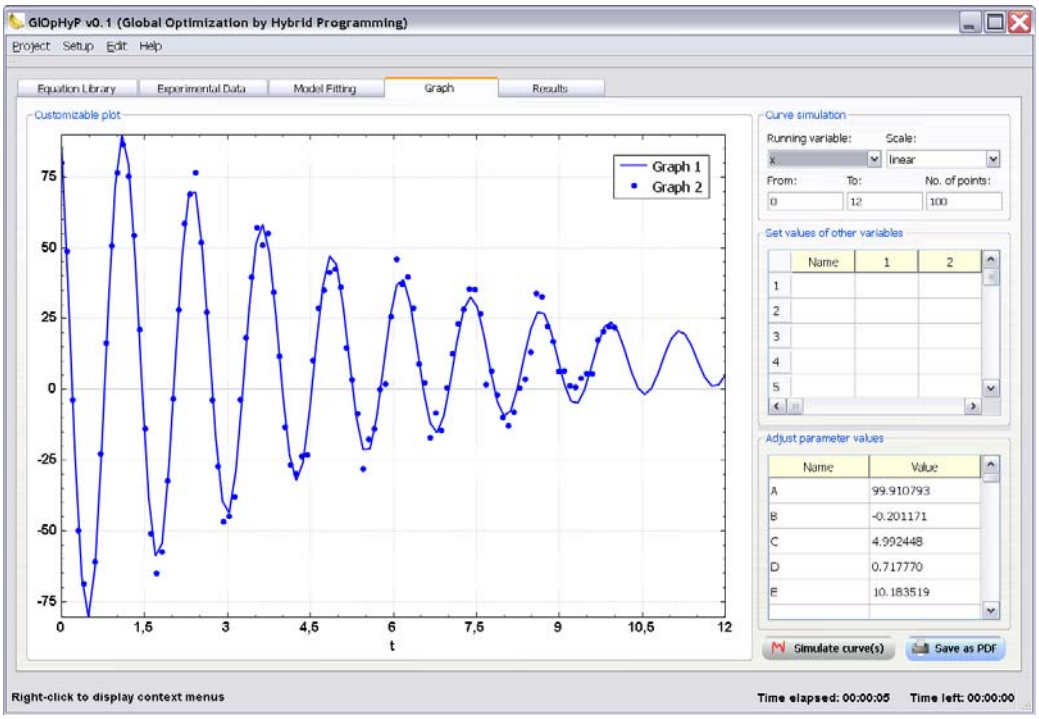
Dans l'exemple ci-dessus nous créons la chaîne `s` en tronquant (`tr`) les caractères données (pour éviter d'y mettre les caractères vides ou ceux qui ne sont pas imprimables), spécifions la police, et affichons le texte en police sélectionnée dans la `statusBar` (via `showMessage`).

On peut faire beaucoup mieux, par exemple mettre un widget (e.g. horloge) dans la barre de statut, grâce à la fonction `addPermanentWidget`. Consultez la documentation pour en savoir d'avantage.

d) utilisation de *plugins*

La fonctionnalité de la bibliothèque Qt, même si très riche, peut parfois s'avérer insuffisante, surtout en cas de besoins spécifiques. Néanmoins, on peut l'accroitre *via* les *plugins*. Il s'agit du code provenant d'autres sources, adapté à l'utilisation dans le cadre du système Qt. Par exemple, il existe une bibliothèque de widgets, `QxtGui`, qui fournit les objets avec les fonctionnalités qui n'ont pas été prises en compte dans le système Qt, comme par exemple l'utilisation du texte riche à l'intérieur des widgets.

Un autre exemple : la création de figures contenant les données scientifiques. Que ce soit un graphe en 2D ou en 3D, les plugins existent pour nous faciliter la vie, en fournissant les nouvelles classes avec les fonctions qui permettent de contrôler la totalité de paramètres. Les mieux connus sont les plugins `QCustomPlot`, `QwtPlot` et `QwtPlot3d`. Ci-dessous se trouve une capture d'écran d'une application qui utilise `QCustomPlot` :



e) interfaçage avec d'autres langues

Des *bindings* existent afin de pouvoir utiliser Qt avec d'autres langages que le C++. Ainsi les langages Java (Qt Jambi), Python (PyQt, PySide, PythonQt), Ruby (QtRuby), Ada (QtAda), C# (Qyoto), Pascal (FreePascal Qt4), Perl (Perl Qt4), Haskell (Qt Haskell), Lua (lqt, QtLua), Dao (DaoQt), Tcl (qtcl), Common Lisp (CommonQt), D (QtD) peuvent être utilisés.

Afin de donner un exemple de facilité avec laquelle cette intégration peut être réalisée, ci-dessous un exemple d'un programme classique "`Hello World`", tout d'abord en C++ et puis en Java et en Python :

C++

```
#include <QApplication>
#include <QPushButton>

int main(int argc, char *argv[])
{
        QApplication app(argc, argv);
        QPushButton hello("Hello world!");
        hello.show();
        return app.exec();
}
```

Java (QtJambi)

```
import com.trolltech.qt.gui.QApplication;
import com.trolltech.qt.gui.QPushButton;

public class HelloWorld
{
        public static void main(String args[])
        {
                QApplication.initialize(args);
                QPushButton hello = new QPushButton("Hello World!");
                hello.show();
                QApplication.exec();
        }
}
```

Python (PyQt)

```
from PyQt4 import QtGui, QtCore
import sys

app = QtGui.QApplication(sys.argv)
hello = QtGui.QPushButton("Hello World!", None)
hello.show()
app.exec_()
```

IV. Projets Qt

Pour améliorer vos connaissances en matière de la création d'interfaces graphiques avec Qt on vous propose la réalisation d'un projet. Vous pouvez travailler indépendamment, ou former une équipe avec vos collègues. Les résultats de votre travail doivent être rendus à la fin du semestre, **avant votre départ en stages**, et comporteront (i) l'archive avec la totalité des fichiers qui font partie du projet, ainsi que le programme exécutable qui marche, et (ii) un rapport d'une dizaine de pages décrivant la nature du projet sélectionné, l'approche adoptée pour résoudre la tâche et les problèmes éventuels rencontrés pendant le travail.

Ci-dessous se trouve la liste des projets au choix. Si vous avez envie d'explorer les fonctionnalités de la bibliothèque `Qt` qui n'ont pas été abordées (ou peu) pendant le cours, comme par exemple les applications multimédia, réseaux ou bases de données, alors choisissez le projet libre.

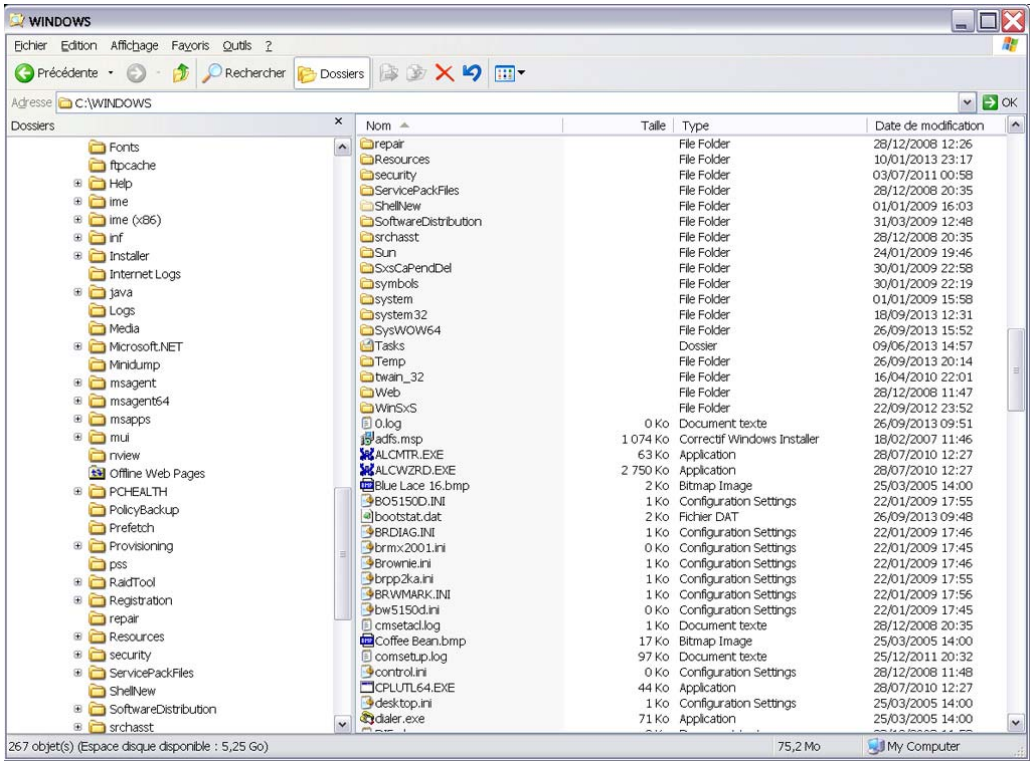
1) encapsulation de(s) ligne(s) de commande(s) sous Linux

Les commandes du système Linux sont nombreuses. Elles sont toutes disponibles à partir d'un terminal, en version lignes de commandes, sous un *shell* tel que bash, csh, tcsh, ksh, zsh, etc. L'objectif de ce projet est de créer une interface graphique entre le système et l'utilisateur, qui permettrait à ce dernier de sélectionner une commande avec ses options, exécuter cette commande et récupérer les résultats en les affichant dans un champ de sortie. Dans la variante la plus simple, l'utilisateur pourrait taper la commande et les options dans un champ d'entrée, mais ceci serait trop peu ambitieux. Pour bien maîtriser la conception d'une interface, il faudra utiliser les widgets comme `Combo Box`, `Check Box`, `Radio Button` et autres pour offrir à l'utilisateur le choix des commandes et de leurs options (on n'est pas obligé d'y inclure la totalité de commandes Linux, ce serait déraisonnable. Contentons-nous de commandes les plus fréquemment utilisées). En autre, il faudra différencier entre la sortie standard et la sortie erreur (si une commande produit un message d'erreur, il faudra réagir d'une certaine façon pour alerter l'utilisateur). Les détails sont à votre charge : montrez votre ingéniosité.

2) explorateur de fichiers

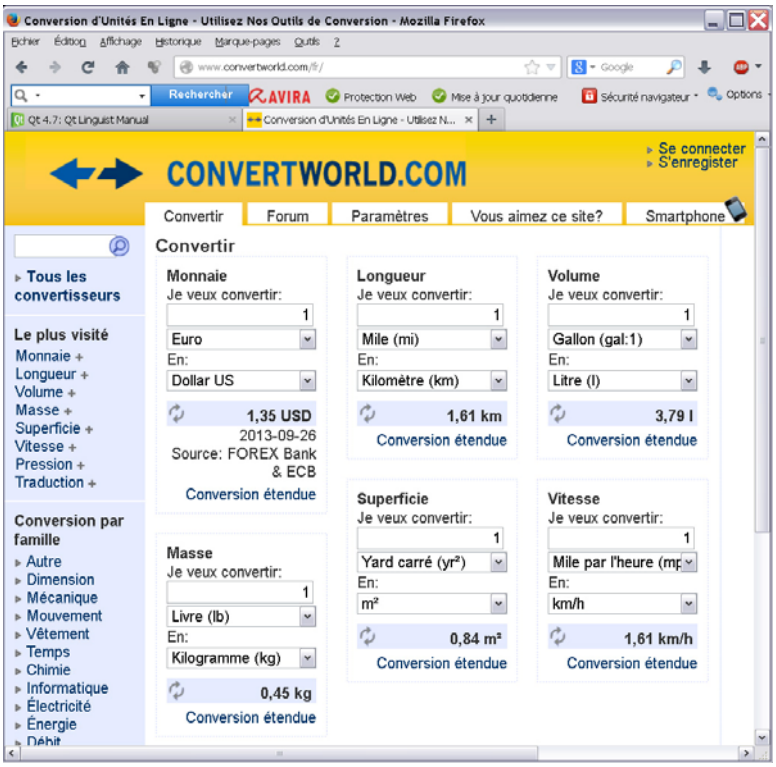
Pour parcourir l'arborescence des fichiers d'un disque dur et pour afficher le contenu des répertoires on peut se servir de la commande "`ls`" dans un terminal, avec ses diverses variantes, mais il serait mieux d'avoir une interface comme celle offerte par l'explorateur de fichiers sous Windows (voir la capture d 'écran ci-dessous). Dans ce projet il ne s'agit pas de recréer la fonctionnalité de l'explorateur (qui d'ailleurs existe aussi sous Linux, e.g. Nautilus), mais plutôt d'offrir à l'utilisateur un jeu restreint des options d'affichage, qui vous permettra de maîtriser l'utilisation de widgets du type `List View`, `Tree View`, `Column View`, etc. Réfléchissez sur la sortie des explorateurs existants : y-a-t-il des fonctionnalités qui manquent ? Ajoutez-les à votre création !

De certain point de vue on peut considérer ce projet comme une variante du Projet 1. Mais l'affichage d'une arborescence avec un mélange de fichiers et de répertoires pose un challenge particulier. Pour cette raison ce problème est traité séparément de la tâche d'affichage de l'ensemble de commandes où le point focal est centré sur la gestion de l'entrée et de sortie.



3) convertisseur des unités (°C/°F, poids, monnaie...)

Dans ce projet on vous propose de créer une interface graphique qui pourrait regrouper l'ensemble de conversions utiles dans la vie quotidienne, un peu comme l'imitation de la fonctionnalité du site Web `convertworld.com` (voir l'image ci-dessous). Evidemment, l'enjeu n'est pas de copier ce que les autres ont déjà fait, mais plutôt de concevoir votre propre interface, avec toutes les améliorations et les extras qui selon vous méritent d'être inclus.



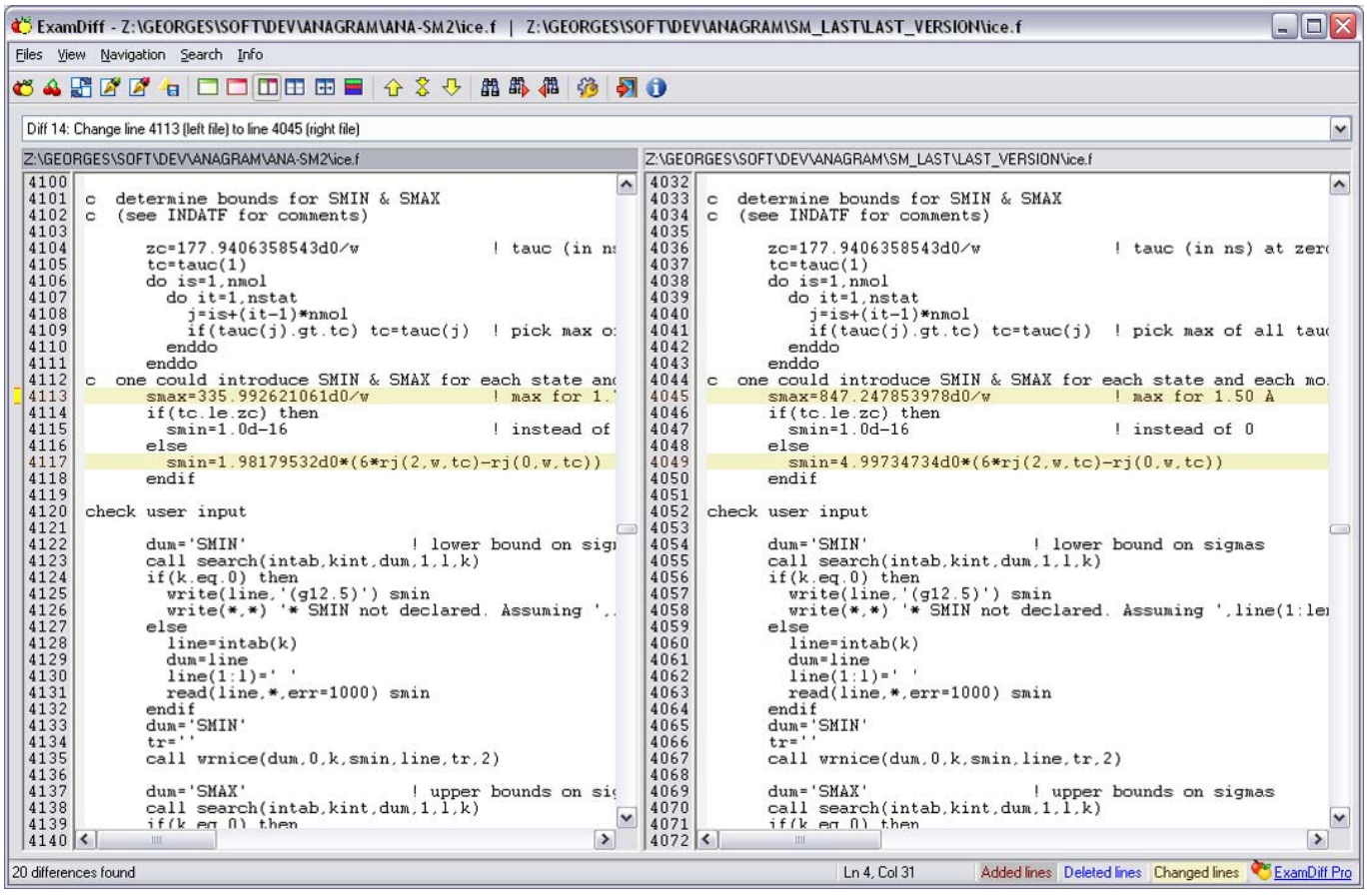
4) trombinoscope

Concevez une variante de trombinoscope permettant de se promener dans une liste de promotions d'une formation (par exemple, le M2I BBS), année par année. Pour chaque année une liste de noms sera affichée. Un clic sur un nom sélectionné affichera la photo de la personne, son CV, les coordonnées, etc.

5) diff

On vous propose de créer une interface graphique qui permettra de comparer deux fichiers texte, en visualisant leurs différences. Il existe une commande Linux "`diff`" qui fait ce qu'on demande, mais uniquement en mode ligne de commande. Ici on voudrait une interface graphique à l'instar du programme `ExamDiff` sous Windows (voir la capture d'écran ci-dessous). Les morceaux identiques de deux fichiers resteront intacts, mais les lignes différentes seront marquées, par exemple, par une couleur différente. Notez bien que la comparaison de deux fichiers texte ressemble beaucoup à l'alignement de deux séquences protéiques : il y a des morceaux identiques, les insertions, les délétions et les gaps. Ici au lieu des acides aminés on a les lignes de texte à comparer. Ceci offre un challenge additionnel : quand deux lignes sont différentes, au lieu de se contenter de les marquer, par exemple en les surlignant, on pourrait marquer plutôt (ou aussi) les caractères différents à l'intérieur de ces lignes. A vous de déterminer les détails de l'interface et de ses fonctionnalités.

Ce projet pourrait être considéré comme une variante du Projet 1, mais la commande `diff` n'est pas comme les autres. Naturellement, vous n'êtes pas obligés à programmer l'alignement de séquences, vous pouvez vous servir de la sortie de `diff`, mais il faudra l'interpréter correctement pour créer l'affichage facile pour l'utilisateur, comme ci-dessous.



6) projet libre (multimédia, réseaux, bases de données…)

C'est ici où votre imagination peut se déchaîner. Si vous avez besoin d'une application particulière, pour vos études ou pour l'utilisation personnelle, vous pouvez la développer dans le cadre du projet libre. La seule exigence imposée est que le degré de difficulté du programme devrait être au moins égal à celui des projets proposés ci-dessus.

V. Références

1) Le premier pas vers la connaissance du système `Qt` est la page Web `http://qt-project.org/`. Vous y trouverez quasiment tout : les logiciels à télécharger, la documentation avec exemples et les forums de discussion concernant tous les aspects d'utilisation de la bibliothèque. Le site est en anglais, mais facile à naviguer.

2) Il y a aussi un site français, `http://qt.developpez.com/`, qui regroupe les sujets concernant `Qt` : la documentation, les questions fréquemment posées (FAQ), forums de discussion, etc. C'est une source très riche en information.

3) Vous pouvez aussi consulter un livre qui a été écrit sur le sujet. Il s'agit de "`C++ GUI Programming with Qt4`" de Jasmin Blanchette et Mark Summerfield. La 1e édition est disponible sur le Web. Deux fichiers contenant ce texte sont joints à l'archive du cours : en format CHM (développé en principe pour Windows) et sa conversion en PDF.

4) Deux tutoriaux basiques sont disponibles dans l'archive du cours. On peut aussi consulter les cours disponibles sur le Web ainsi que les vidéos présentant les divers aspects de la programmation. Certains de ces tutoriels concernent les versions précédentes de Qt, mais les principes ne changent pas et les différences sont négligeables si on reste au niveau de la même version majeure (ici : 4.x). Voici une sélection des adresses Web :

- http://fr.openclassrooms.com/informatique/cours/programmez-avec-le-langage-c/introduction-a-qt
- http://www.zetcode.com/gui/qt4/
- http://www.digitalfanatics.org/projects/qt_tutorial/fr/
- http://www.youtube.com/user/VoidRealms (il y a des vidéos sur Qt vers bas de la page)
- http://doc.qt.digia.com/4.3/tutorial.html
- http://www.youtube.com/playlist?list=PL2D1942A4688E9D63 (toute une série de vidéos)

Il y a beaucoup d'autres sites, offrant l'aide plus ou moins pointue, en fonction de besoins spécifiques. N'hésitez pas d'utiliser les moteurs de recherche.

5) Pour compléter l'information fournie, vous trouverez dans l'archive joint le cours de C++ qui faisait partie de l'enseignement du M2P Bioinformatique il y a quelques années, créé par E. Courcelle. Il peut être utile pour éclairer quelques points obscures du langage C++ si vous tombez sur en exercice difficile. Deux versions sont incluses : HTML avec toute l'arborescence de fichiers, et la version PDF.