TP 8 Initiation au développement d'interfaces graphiques avec Qt

1. Première application Qt : Calcul

QApplication, QWidget, QPushButton, QLabel, QComboBox, QLineEdit, Slots et signaux.

Pour créer le projet QtCreator de cette application, créer comme toujours une application Non-Qt project / Plain C++ Application / CMake, et prendre exemple sur le fichier `CMakeLists.txt` vu en cours pour activer la compilation avec Qt en plus du standard C++ 17. Dans cet exercice, nous n'utiliserons pas les layouts de Qt et positionnerons les widgets directement avec des coordonnées et tailles (méthode `setGeometry`). Comme il s'agit du premier exercice sur Qt, l'intégralité des classes, méthodes, signaux, slots à utiliser sont précisés, mais n'hésitez pas à consulter la documentation de Qt dont un point d'accès important est http://doc.qt.io/qt-5/qtwidgets-module.html (documentation de toutes les classes widgets) pour vous familiariser avec la structuration de cette documentation, notamment pour ce qui concerne la description des méthodes, slots et signaux, et parce que travailler avec la documentation est tout à fait normal, il est impossible de connaître toutes les méthodes de toutes les classes de Qt !

1. Déclarer une classe `calcul`, sous classe de `QWidget`[1] formée des attributs dont les noms et types suivent : `_operande1` (`QLineEdit`), `_operande2` (`QLineEdit`), `_operateur` (`QComboBox`), `_resultat` (`QLabel`), `_calculer` (`QPushButton`), `_quitter` (`QPushButton`). Le constructeur de `QComboBox` construit un widget vide, et il faut appeler `addItem` pour rajouter les éléments, vous ajouterez 4 éléments à `_operateur` : `+`, `-`, `*`, `/` et vous positionnerez les widgets dans la fenêtre `calcul` (de taille 450 sur 50) pour obtenir l'apparence ci-contre (peu importe si ce n'est pas « harmonieux », si les widgets sont trop « hauts », on fera plus tard des « layouts » plus propres).

2. Écrire le programme principal qui instancie la fenêtre `calcul` et se termine quand la fenêtre est fermée.
3. *On va maintenant gérer le bouton `_quitter` afin qu'il quitte effectivement l'application, ou plutôt qu'il ferme la fenêtre (et si cette fenêtre est la dernière de l'application, l'application se termine automatiquement).*
   Le code que nous allons écrire réside dans le constructeur de calcul, il ne connaît donc pas la `QApplication` déclarée dans le `main`, il est donc malaisé de se connecter au slot `QApplication::quit` (comme dans l'exemple vu en cours). Nous allons donc utiliser le slot `close` de `QWidget` (hérité dans `calcul`). Dans le constructeur de `calcul`, connecter le signal `QPushButton::clicked` de `_quitter` au slot `calcul::close` de l'objet courant afin qu'un clic sur le bouton provoque une fermeture de la fenêtre (et donc de l'application).
4. *Passons au bouton `_calculer`.*
   Commencer par définir un slot `oncliccalculer` (sans arguments) et connecter ce slot au signal `QPushButton::clicked` de `_calculer`. Dans cette méthode, on doit faire le calcul `_operande1 _operateur _operande2`. La méthode `text` de `QLineEdit` retourne le texte saisi sous la forme d'une `QString` qu'il faut convertir en `std::string` avec `toStdString`. `std::stof` convertit une `std::string` en `float`. Écrire dans un premier temps le code qui définit deux variables de type

1 Dans cet exercice, nous n'allons pas **émettre** de signaux mais simplement définir des slots et connecter des signaux existants à ces slots, donc il n'est pas indispensable d'utiliser la macro `Q_OBJECT` dans la déclaration de la classe. Cependant, il peut être une bonne idée de l'utiliser systématiquement dès qu'on définit une classe d'interface graphique, si on veut rajouter par la suite l'émission de signaux (et parce que son rôle n'est pas uniquement de permettre l'émission de signaux).

`float` à partir des valeurs saisies par l'utilisateur et les affiche sur la sortie standard. Attention `std::stof` ne fonctionne pas (en fait, lève une exception) si la conversion ne peut pas être faite (chaîne vide, chaîne contenant autre chose qu'un flottant). Nous ne prendrons en compte que le cas de la chaîne vide : faire en sorte qu'une chaîne vide dans un widget opérande soit considérée comme la valeur 0.

5. *Il faut maintenant faire le calcul, et pour cela récupérer le signe choisi dans `_operateur`.*
   La méthode `currentText` de `QComboBox` retourne le texte actuellement sélectionné. Faire le calcul en fonction de l'opérateur choisi dans `_operator`. Le résultat du calcul est donc un `float`. Ce `float` peut être converti en `std::string` par `std::to_string`, à son tour convertie en `QString` par `QString::fromStdString`. La `QString` ainsi obtenue est utilisée pour changer la valeur affichée par `_resultat` par un appel à `setText`.
6. *On veut maintenant que le résultat du calcul soit mis à jour dès que la valeur d'un des opérandes change ou qu'un nouvel opérateur est choisi par l'utilisateur. Une fois que ceci sera fait, le bouton `_calculer` deviendra d'ailleurs totalement inutile.*
   `QComboBox` a un signal `currentTextChanged` qui est émis quand le choix de l'utilisateur est changé. Un `QLineEdit` émet un signal `textChanged` quand le texte saisi est édité. Ces deux signaux ont un paramètre qui est une `QString` contenant la nouvelle valeur du texte du widget. Ils devraient dont être connectés à un slot qui a cette signature. Toutefois, Qt permet de connecter un signal à un slot qui a une signature plus courte à condition qu'il y ait compatibilité sur les premiers arguments de la signature : les arguments supplémentaires sont alors « ignorés ». Ceci devrait vous permettre de faire très facilement une mise à jour du résultat affiché après chaque action de l'utilisateur.

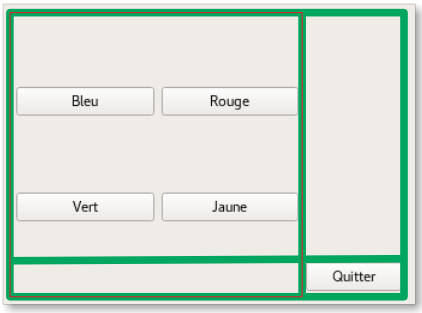
2. Séquence de couleurs (5/5) : Interface graphique de jeu « Simon »

QGridLayout, Feuille de style Qt, QPushButton, QLineEdit, QLabel, QMessageBox, std::map.

Attention : pour faire cet exercice, il faut avoir vu la totalité du chapitre 8 du cours, n'essayez pas de le faire alors que nous n'avons pas fini ce chapitre.
Dans cet exercice, nous allons réaliser une interface graphique permettant de saisir une séquence de couleurs et aller jusqu'au développement d'un jeu à deux joueurs dans lequel chaque joueur doit, à tour de rôle, reproduire la même séquence de couleur et ajouter une nouvelle couleur. Vous pouvez donc récupérer le code de la classe écrite dans la partie 4 de cette suite d'exercices. Vous aurez besoin de consulter la documentation de Qt, car nous allons utiliser certaines méthodes qui n'ont pas été vues en cours. Les liens sont donnés dans l'exercice, mais n'hésitez pas à consulter d'autres pages.

1. Avant de commencer l'interface graphique, nous allons améliorer un peu la classe `sequence` en fonction de ce que nous avons appris depuis le tp 2, afin de simplifier la mise au point de l'application que nous allons réaliser à partir de la question 2 :
   Faire en sorte qu'une `couleur` puisse être envoyée sur un flux de sortie (soit c une `couleur`, on veut pouvoir écrire `std::cout << c;`) et affiche bleu, jaune, etc. : soit, pa
   Faire en sorte qu'une `sequence` puisse être envoyée sur un flux de sortie.
   Supprimer la méthode `copier` et faire en sorte que la copie d'une séquence puisse être faite par un appel au constructeur par recopie afin de pouvoir écrire, par exemple, soit `s1` une `sequence`, `sequence s2(s1);`.
   Définir un opérateur d'affectation afin de pouvoir écrire, par exemple, soit `s1` et `s2` deux `sequence` : `s2 = s1;`.
   Supprimer la méthode `comparer` et faire en sorte que la comparaison se fasse par l'opérateur ==.
   Supprimer la méthode `acces` et faire en sorte que l'accès à un élément de la séquence soit fait avec l'opérateur `[]`.
   Écrire une méthode `vider` qui vide la séquence.

2. Nous allons écrire dans cette question une classe `simon`, sous-classe de `QWidget` représentant la fenêtre ci-contre (sans traits de couleur), en plaçant les widgets progressivement.



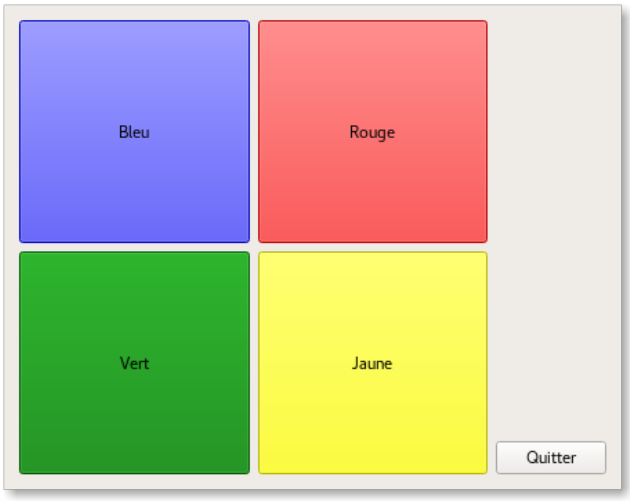
Dans cette fenêtre, les widgets seront créés (dans le constructeur) et rangés à l'aide d'un (premier) `QGridLayout` de deux colonnes et deux lignes qu'on appellera `layoutgeneral` et représenté par des lignes vertes sur la capture d'écran ci-contre. Ce `QGridLayout` contiendra uniquement le bouton Quitter, les boutons de couleurs seront placés plus tard.

Nous allons simplement placer le bouton Quitter pour commencer, sur la deuxième colonne, deuxième ligne : cette deuxième colonne et deuxième ligne seront de largeur / hauteur fixées : la taille du bouton ne changera pas alors que la taille de la zone contenant les 4 couleurs, sur la première colonne, et occupant les 2 lignes sera variable.

On va maintenant créer et ranger les 4 boutons de couleurs. Pour cela on va créer un deuxième `QGridLayout` appelé `layoutcouleurs` (de taille 2 sur 2) qui sera rangé dans la case 0,0 du `layoutgeneral`, et étendu sur deux lignes : sur la figure, ce layout est repéré par le cadre rouge : remarquer qu'il occupe les 2 lignes du `layoutgeneral` (sur la première colonne). Pour faire cela, on appellera la méthode `QGridLayout::addLayout` (qui s'utilise comme `addWidget` mais prend comme paramètre un `QLayout` http://doc.qt.io/qt-5/qgridlayout.html) sur le `layoutgeneral` en lui passant en paramètre le `layoutcouleurs`, utiliser les arguments 4 et 5 de la méthode pour faire en sorte que le `layoutcouleurs` occupe les 2 lignes de la première colonne du `layoutgeneral`.

Les 4 boutons de couleurs seront évidemment ajoutés au `layoutcouleurs`.

3. On va maintenant ajouter de la couleur. Qt permet de définir l'apparence des widgets à l'aide de feuilles de style dont la syntaxe est fortement inspirée des CSS. http://doc.qt.io/qt-5/stylesheet-syntax.html .



Ainsi pour fixer la couleur de fond des `QPushButton` à rouge, on utilisera la feuille de style « `QPushButton { background-color: blue; }` »

Une feuille de style, représentée sous la forme d'une chaîne de caractères est associée à un widget à l'aide de la méthode `setStyleSheet`. Colorer les 4 boutons.

`QSizePolicy` http://doc.qt.io/qt-5/qsizepolicy.html décrit la façon dont un widget est redimensionné. Par défaut, un `QPushButton` se redimensionne en largeur mais pas en hauteur, ce qui explique le rendu actuel. C'est un choix raisonnable dans une interface graphique : tous les boutons ont habituellement la même hauteur. Ce comportement doit donc être changé pour les 4 boutons de couleur : `QWidget::setSizePolicy` permet de choisir le mode de redimensionnement horizontal (premier paramètre) et vertical (deuxième paramètre). `QSizePolicy::Expanding` permet de signaler un agrandissement maximum. Faire en sorte que les boutons de couleurs se redimensionnent comme sur l'image ci-contre.

4. Déclarer un attribut de type `std::map` dans `simon` permettant, à partir d'une `couleur` (clé), d'accéder au `QPushButton*` (valeur) représentant la `couleur`, ce `map` étant évidemment rempli par le constructeur de `simon`.

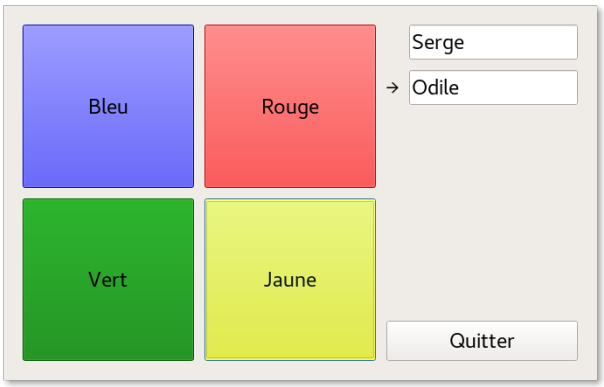
   Écrire une méthode privée `boutonverscouleur` qui à partir d'un `QPushButton*` retourne la `couleur` correspondante (en cherchant dans le `map`).

5. Déclarer un slot `onclicquitter` connecté au clic sur le bouton quitter. Ce slot fera apparaître une boîte de dialogue *modale* (qui interrompt le fonctionnement de la fenêtre principale tant que cette boîte de dialogue n'est pas fermée) demandant à l'utilisateur s'il veut effectivement quitter. Qt offre

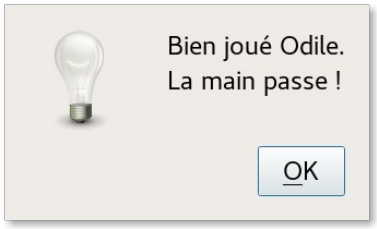
une gestion simplifiée de boîtes de dialogue simples à travers la classe `QMessageBox` http://doc.qt.io/qt-5/qmessagebox.html#details . Pour des boîtes très simples comme celle qu'on veut écrire ici, le plus rapide est d'utiliser les méthodes de classe de `QMessageBox` : `information`, `warning`, `question`, `critical`. Ici, on utilisera `QMessageBox::question` qui prend comme paramètre le pointeur sur le widget parent (la fenêtre principale), le titre de la boîte, le message de la boîte, et une combinaison (par l'opérateur binaire |) de boutons (voir `QMessageBox::StandardButton`) et retourne le bouton choisi. Si l'utilisateur clique sur oui, la fenêtre courante sera fermée. Le texte des boutons devrait apparaître en anglais, car nous n'avons pas configuré l'application pour utiliser la langue par défaut de l'utilisateur, et dans ce cas, Qt affiche des messages en anglais. Pour utiliser la langue par défaut de l'utilisateur, dans le main, vous initialiserez l'application de cette façon :


```
QTranslator qtTranslator;
qtTranslator.load("qt_"+QLocale::system().name(),QLibraryInfo::location(QLibrar
yInfo::TranslationsPath));
QApplication app(argc, argv);
app.installTranslator(&qtTranslator);
```

6. Déclarer un slot `oncliccouleur` et connecter ce (seul) slot au signal clic sur les 4 boutons de couleurs. Dans le code d'un slot, la méthode `sender` http://doc.qt.io/qt-5/qobject.html#sender retourne l'objet à l'origine du signal. Ainsi, dans le slot `oncliccouleur` on pourra accéder au bouton cliqué. À l'aide de la méthode `boutonverscouleur`, on pourra retrouver la valeur correspondante. Même si `sender`() retourne un `QObject`, il est certain, par construction du programme, que le `sender` est un `QPushButton` (puisque le slot `oncliccouleur` n'est connecté qu'à des `QPushButton`), donc la valeur retournée par `sender` pourra être convertie sans risque en `QPushButton*`.
   Déclarer une `sequence` comme attribut de `simon` et faire en sorte qu'un clic sur un bouton de couleurs ajoute la couleur correspondante à la séquence.
   Afficher la séquence sur la sortie standard à chaque ajout afin de vérifier que le comportement est bien le comportement attendu.
7. On veut maintenant programmer le jeu à proprement parler dans lequel il s'agit pour un joueur de reproduire (de mémoire) la séquence : on déclarera donc un attribut supplémentaire : l'indice dans la séquence de la prochaine couleur devant être cliquée. Si on clique sur une mauvaise couleur, un `QMessageBox` sera affiché pour informer que le joueur a perdu. Quand la séquence est correctement reproduite en entier, l'utilisateur cliquera sur une couleur qui s'ajoutera à la séquence, et un `QMessageBox` sera affiché pour informer qu'on doit recommencer la séquence depuis le début (avec la couleur supplémentaire ajoutée). Évidemment, au début du programme, la séquence est vide : la première couleur cliquée est la première ajoutée à la séquence, et immédiatement le `QMessageBox` informant qu'on recommence est donc affiché.
8. Modifier le programme afin de pouvoir choisir la séquence de couleurs initiale : si le programme est lancé en lui passant en arguments de ligne de commande (`argc/argv`) ces arguments sont considérés comme des couleurs (séparées par des espaces). Lancé sans argument, le comportement est celui de la question précédente, et commence par une séquence vide.
9. Modifier le déroulement du jeu pour en faire un jeu à deux joueurs :
   Les joueurs peuvent saisir leur nom (à tout moment du jeu, et les noms par défaut sont Joueur 1 et Joueur 2) et une flèche indique quel joueur joue. À tour de rôle, le joueur actif doit reproduire la séquence courante, suivie d'une nouvelle couleur de son choix. Une fois que ceci est fait, la main passe et c'est à l'autre joueur de faire la même chose.



Il vous faut bien évidemment réorganiser un peu le `layoutgeneral` pour intégrer les `QLineEdit` (noms des joueurs) et `QLabel` (signalant le joueur courant) : deux lignes de plus, deux colonnes de plus. La flèche est un simple caractère unicode dans un `QLabel` : pour l'afficher dans un `QLabel`, vous utiliserez `setText("\u2192")`. L'autre `QLabel` contiendra un espace, et les textes de ces 2 `QLabel` devront être mis à jour à chaque fois que la main passe.
En cas d'erreur, la boîte de dialogue ci-dessous (à gauche) sera affichée, et précisera le nom du joueur perdant et la longueur de séquence atteinte.



Quand la séquence suivie de la nouvelle couleur seront saisies, la boîte de dialogue ci-dessous (à droite) sera affichée.