TP 1
Prise en main de Qt

Objectifs : Configuration de l'environnement de développement. Compilation d'un premier programme. Modification d'un code fourni.

Durée : 1 séance

Qt est une bibliothèque C++ permettant la programmation d'interfaces graphiques portables. Vos systèmes d'exploitation respectifs, pour lesquels vous allez développer au cours de ces TP, en sont une bonne illustration. Le but de ce premier TP est de compiler un programme fourni puis d'y apporter quelques modifications.

1 Compilation d'un projet Qt

1.1 Présentation

La compilation d'un programme basique utilisant Qt peut parfois se faire par la simple commande :

```
g++ -I/usr/include/qt5/QtCore -o main main.cpp -L/usr/lib -lQtGui -lQtCore
```

Cependant, pour un projet classique, la compilation nécessite l'utilisation d'un outil supplémentaire : le programme `moc` (*Meta Object Compiler*). Celui-ci analyse certains fichiers d'en-tête (`.h`) de votre programme à la recherche des extensions au langage C++ que Qt apporte. En deux mots, Qt définit en plus des mots clefs « public », « protected » et « private » du C++ les mots clefs « signals » et « slots » (dont nous reparlerons en cours). Le programme `moc` construit, à partir des déclarations que ces mots clefs précèdent, du code C++ supplémentaire qu'il place dans un fichier `moc_XXXXX.cpp` qu'il faut compiler et lier au programme final. La figure 1 illustre un exemple de compilation d'un projet constitué d'un programme principal (`main.cpp`) et d'un fichier séparé (`MyWidget.{h,cpp}`) qui définit un widget personnalisé.

Pour simplifier la compilation, Qt fournit un outil capable de générer le Makefile approprié pour un projet et une architecture donnés : `qmake`. Ce programme utilise un fichier (tel `main.pro`, listing 1) décrivant un projet comme celui illustré par la figure 1. Notez que depuis la version 6 de Qt sortie en décembre 2020, CMake est venu remplacer par défaut qmake (qui reste toutefois disponible).

```
TEMPLATE = app
 CONFIG += warn_on release
     QT += core gui widgets
 HEADERS = MyWidget.h
 SOURCES = MyWidget.cpp main.cpp
  TARGET = main
```

Listing 1 – Fichier main.pro

Le Makefile est alors généré grâce à la commande :

```
e102pc01$ qmake [main.pro]
```

Si un seul fichier `.pro` est présent dans le dossier courant, la commande `qmake` seule suffit.

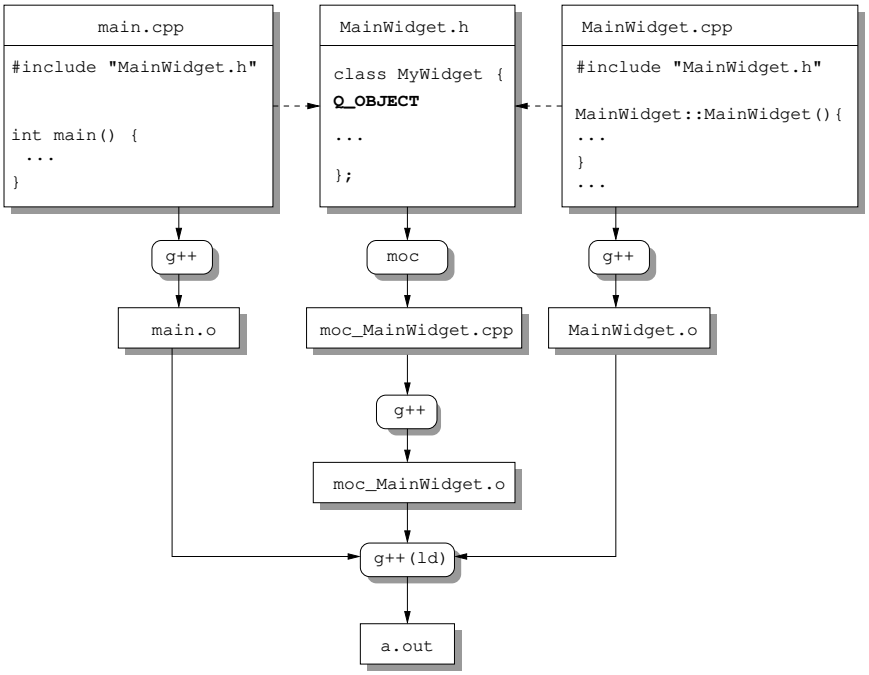

FIGURE 1 – Compilation d'un projet Qt

1.2 Récupération et compilation d'un exemple

1.2.1 Téléchargez les fichiers sources du TP sur la page Moodle du cours.

À savoir :

`main.pro`	Description du projet pour qmake
`main.cpp`	Programme principal
`MainWidget.{h,cpp}`	Widget de la fenêtre principale de l'application
`ColorWidget.{h,cpp}`	Un widget simple affichant une couleur RGB

1.2.2 Ouvrez le fichier `main.pro` et parcourez son contenu.

1.2.3 Nous allons ici nous convaincre qu'il vaut mieux laisser à Qt/qmake le soin de produire un fichier Makefile à notre place.

Ouvrez le fichier `main.pro` avec QtCreator [1]. À la première question qui vous est posée lors de l'ouverture du projet, si vous avez installé Qt sous Windows via MSYS2 choisissez le kit « Desktop Qt MinGW-w64 64bit (MSYS2) » et dans tous les cas, vérifiez que les modes de compilation "Debug" et "Release" sont bien cochés dans les "Details..." du kit. QtCreator permet en effet de « déverminer » votre programme au sein de l'IDE.

Ensuite, lancez qmake (Menu "Compiler > Exécuter qmake"). Le Makefile devrait alors se trouver dans un dossier nommé "build-*" situé au même niveau que le dossier racine de votre code source. Ouvrez-le et parcourez rapidement son contenu.

1.2.4 Lancez la compilation ("Compiler > Compiler le projet" ou Ctrl+B) et exécutez le programme obtenu (Ctrl+R).

1. Commande `qtcreator` sous Linux.

2 Quelques retouches

Vous allez dans cette section faire vos premiers pas avec la bibliothèque en modifiant le code téléchargé précédemment. Attention, dans ce code certains widgets sont positionnés et dimensionnés avec des valeurs absolues en nombre de pixels. Il s'agit bien sûr d'une très mauvaise idée car vous pourriez êtres amenés à modifier ces valeurs par tâtonnement. Dans les prochains TP des gestionnaires de disposition seront bien sûr utilisés.

Conseil QtCreator permet d'accéder facilement à la documentation d'une classe via la touche F1 lorsque le curseur est positionné sur un nom de méthode, un type ou une variable. Vous pouvez aussi trouver sur le site `qt.io` la documentation de toutes les classes.

2.1 Ajustez si besoin les positions des différents widgets (méthodes `setGeometry`).

2.2 En utilisant la documentation de la classe `QWidget` dont dérive la classe `MainWidget` du programme fourni, modifiez le titre (*Window Title*) de la fenêtre de l'application.

2.3 Le champ de saisie (`QLineEdit`), dans lequel s'affichent les 3 composantes de la couleur actuelle du `ColorWidget`, est créé et initialisé dans le constructeur de la classe `MainWidget`. Faites en sorte que le champ ne soit pas modifiable (c.-à-d. « *read-only* »).

2.4 Ajoutez un bouton (`QPushButton`) permettant de quitter l'application. Pour cela, connectez le signal `QPushButton::clicked` de ce nouveau bouton au slot `MainWidget::quitPressed` déjà présent. (Voir le support de cours mais aussi les appels à `connect` déjà présents.)

2.5 Ajoutez un second champ de saisie, non éditable, au dessus du champ déjà existant. Il sera utilisé dans la question suivante pour afficher la position de la souris.

2.6 On veut maintenant prendre en compte dans le widget principal des évènements liés au déplacement de la souris. Pour cela, il suffit de redéfinir dans la classe `MainWidget` la méthode (virtuelle et protected)

```
virtual void QWidget::mouseMoveEvent(QMouseEvent * e);
```

de la classe `QWidget`. Cette méthode est appelée automatiquement lorsque la souris est déplacée alors qu'au moins un bouton est appuyé.

Quand la souris est déplacée dans le widget principal, ses coordonnées relativement au `MainWidget` devront être affichées sous la forme "(x,y)" dans le champ de saisie créé en (e). Les coordonnées seront affichées avec un alignement (`QLineEdit::setAlignment()`) à gauche, centré ou à droite selon le bouton de souris qui est appuyé. Cette dernière information, tout comme la position, est disponible dans le `QMouseEvent` et doit être comparée avec des constantes comme `Qt::LeftButton`. Vous pourrez utiliser la méthode `QString::arg()` pour mettre en forme la chaîne "(x,y)".

Attention : la méthode `QMouseEvent::buttons()`, au pluriel, retourne une combinaison de bits associés aux boutons qui sont enfoncés. L'usage d'opérations bit à bit peut être nécessaire.

Notez au passage l'usage « intensif » fait par Qt de constantes de type `enum` aux noms explicites (noms des boutons, types d'alignements, etc.).

3 Bonus pour les plus avancés

3.1 À l'aide d'un objet de type `QTimer` capable d'émettre un signal de façon périodique, faites afficher dans un objet `QTimeEdit` l'heure courante (donnée par une méthode statique de la classe `QTime`).

3.2 Ajoutez les widgets nécessaires pour offrir un chronomètre (Start, Stop, Clear) à l'aide d'un second timer et d'un objet `QElapsedTimer`.

La méthode statique `QTime::fromMSecsSinceStartOfDay` et la méthode `QTime::toString` peuvent vous être utiles pour convertir un nombre de millisecondes en une chaîne de type heures:minutes:secondes.

Seul un bouton devra être actif à un instant donné.