Programmation orientée objet avancée
TP/Projet Snake

1 Prise en main

1.1 Configuration

Suivez la section 1.1.1, 1.1.2 ou 1.1.3 en fonction de votre système d'exploitation.

1.1.1 Utilisateurs Windows

Téléchargez les deux fichiers suivants à partir de Celene :
- **Qt 5.15.2 (version light) (mingw64)**
- **Snake - Qt 5.15 (mingw64)**

Décompressez les deux archives (en sélectionnant l'option "Extraire ici"), de manière à avoir dans un même dossier les deux dossiers **qtbase-small-5.15.2-x64** et **snake-with-qt-5.15**. L'arborescence des dossiers doit être
- `snake-with-qt-5.15`
  - data
  - ... (fichiers .cpp, .hpp, Makefile...)
- `qtbase-small-5.15.2-x64`
  - `include`
  - `lib`
  - `plugins`
  - `src`

L'archive **qtbase-small-5.15.2-x64** contient une version de la bibliothèque Qt préalablement allégée et compilée par l'enseignant avec MinGW-w64. Cela permet une mise en place et une configuration rapide du projet (entre autres inconvénients, la version compilée disponible sur `https://www.qt.io/` est beaucoup plus volumineuse. Elle nécessite une installation et donc les droits administrateur). La bibliothèque Qt est décrite plus en détail en section 2.

Vous pouvez travailler
- soit avec Code::Blocks. Dans ce cas, ouvrez le fichier de projet **snake_win.cbp**.
- soit en ligne de commande. Dans ce cas, renommez le fichier **Makefile_win** en **Makefile**. La compilation et la génération de l'exécutable seront effectuées en entrant la commande `mingw32-make`, qui interprète les règles de compilation/édition des liens contenues dans le fichier **Makefile**.

Si vous exécutez en ligne de commande, vous devez copier les trois fichiers **.dll**, depuis **qtbase-small-5.15.2-x64/lib**, dans le dossier du projet Snake. Vous devez également copier le dossier **platforms**, depuis **qtbase-small-5.15.2-x64/plugins**, dans le dossier du projet Snake

1.1.2 Utilisateurs Linux

Si vous utilisez votre machine personnelle sous Linux, la bibliothèque Qt doit être installée. Pour ce faire, entrez la commande suivante dans un terminal :

```
sudo apt install qtbase5-dev
```

Téléchargez uniquement **Snake - Qt 5.15 (mingw64)** à partir de Celene. Vous pouvez travailler

- soit avec Code::Blocks. Dans ce cas, ouvrez le fichier de projet **snake_linux.cbp**.
- soit en ligne de commande. Dans ce cas, renommez le fichier **Makefile_linux** en **Makefile**. La compilation et la génération de l'exécutable seront effectuées en entrant la commande `make`, qui interprète les règles de compilation/édition des liens contenues dans le fichier **Makefile**.

1.1.3 Utilisateurs Mac

Si vous utilisez un Mac, la bibliothèque Qt doit être installée (voir les documents dans la section TP/Projet sur Mac sur Celene). Vous compilerez en ligne de commande. Renommez le fichier **Makefile_macos** en **Makefile**. La compilation et la génération de l'exécutable seront effectuées en entrant la commande `make`, qui interprète les règles de compilation/édition des liens contenues dans le fichier **Makefile**

1.2 Première compilation et exécution

Compilez le projet et exécutez-le. Vous pouvez changer la direction du serpent avec les touches flèches. Si le projet ne compile pas où si la fenêtre du Snake ne s'affiche pas, c'est que vous n'avez sans doute pas respecté l'organisation des dossiers donnée précédemment, ou que le chemin vers le compilateur n'est pas correct.

1.3 Quelques informations sur les bibliothèques et la configuration d'un projet

Nous expliquons ici comment le projet est configuré. Lorsque vous "compilez" un projet. Sont réalisées, entre autres,

- l'étape de compilation proprement dite, qui génère un fichier objet (d'extension .o) par fichier source .c ou .cpp (si le code contenu dans ce fichier est correct). Un fichier objet contient la traduction du code source en code machine. Celui-ci est dépendant du système d'exploitation, de l'architecture et du compilateur. Le code machine est le langage directement interprétable par le processeur.
- l'étape d'édition des liens (si tous les fichiers .c/.cpp du projet ont pu être compilés). L'éditeur de liens, ou linker, construit l'exécutable final en combinant les fichiers objet et les bibliothèques. On parle ici de bibliothèques statiques (d'extension .a). Dans le cas de la bibliothèque standard du C++, cet ajout a lieu mais est complètement transparent pour le programmeur.

Dans Code::Blocks, les options du compilateur et de l'éditeur de liens pour le projet en cours se trouvent dans le menu "Project" > item "Build options...". Dans l'onglet "Search directories", vous verrez les chemins relatifs vers les headers (.hpp) et les bibliothèques (libXXX.a) (veillez à ce que la racine "snake" soit sélectionnée dans l'arborescence de gauche, pas "Debug" ou "Release").
Dans l'onglet "Linker settings", des commandes sont ajoutées pour indiquer à l'éditeur de liens qu'il doit ajouter des bibliothèques lors de la génération de l'exécutable (il va chercher ces fichiers dans les dossiers spécifiés dans "Search directories" > "Linker").

2 La bibliothèque Qt

Qt (prononcez "cute") est une bibliothèque C++ pour concevoir des applications avec interfaces graphiques (tout comme GTK, SDL, wxWidget, *etc*). Elle est open-source et portable, un code utilisant Qt peut donc être compilé sur différentes architectures et systèmes d'exploitation, sur des ordinateurs de bureaux ou sur mobiles.
`https://www.qt.io/`
`https://doc.qt.io/qt-5/index.html`

2.1 Classe `QObject`

La classe `QObject` est la classe mère de tous les objets Qt. Les objets Qt sont organisés sous forme d'arborescences d'objets. Lorsqu'un `QObject` est créé avec un autre objet en tant que parent, l'objet s'ajoute automatiquement à la liste `children()` du parent[1]. Le parent prend possession de l'objet ; c'est-à-dire qu'il supprimera automatiquement ses enfants dans son destructeur.

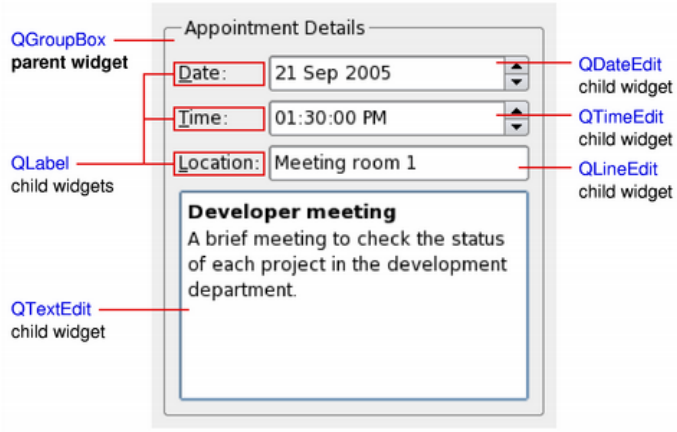
Le constructeur de recopie et l'opérateur d'affectation de `QObject` sont privés, ce qui les empêche d'être appelés de l'extérieur. Par conséquent, on utilisera systématiquement des pointeurs sur `QObject` (ou sur une sous-classe de `QObject`) là où l'on pourrait être tenté d'utiliser un `QObject` comme valeur. Par exemple, sans constructeur de recopie, il est impossible d'utiliser un objet sous-classe de `QObject` comme valeur à stocker un conteneur `vector` ou `list`. On doit stocker des pointeurs.

2.2 Classe `QWidget`

La classe `QWidget` est la classe de base de tous les objets d'interface graphique : fenêtres, boutons, menus, zones de saisie, ...

Le widget est l'élément de base de l'interface graphique. Il occupe une certaine zone rectangulaire à l'écran. Il peut s'afficher et recevoir des évènements (souris, clavier, ...). Un widget peut également avoir un widget parent, dans lequel il est intégré. Un widget qui n'est pas intégré dans un widget parent est une fenêtre. `QMainWindow`, `QFrame` et les différentes classes fille de `QDialog` sont les types de fenêtres les plus courants. Un widget possédant un parent (intégré dans une fenêtre, donc) sera la plupart du temps un contrôle : bouton, zone de saisie, liste déroulante, ...

Un widget parent peut avoir un nombre quelconque d'enfant(s). L'exemple ci-dessous montre un widget `QGroupBox` utilisé pour contenir divers widgets enfants dans une disposition fournie par `QGridLayout` :



Le constructeur de `QWidget` reçoit deux paramètres optionnels :

- `QWidget * parent = nullptr` est le parent du nouveau widget. Si c'est `nullptr` (la valeur par défaut), le nouveau widget sera une fenêtre. Sinon, il sera intégré dans un widget parent.
- `Qt :: WindowFlags f = 0` (si disponible) définit les flags de la fenêtre ; la valeur par défaut convient à presque tous les widgets, mais pour obtenir, par exemple, une fenêtre sans cadre de fenêtre, vous devez utiliser des flags spéciaux.

Chaque widget effectue toutes les opérations d'affichage à partir de sa méthode `paintEvent()`. Celle-ci est appelée automatiquement chaque fois que l'affichage du widget doit être rafraîchi. Comme de nombreuses méthodes gérant les événements, elle peut être redéfinie. Dans le projet qui vous est fourni,

1. Attention, les notions de parent et d'enfant n'ont rien à voir avec l'héritage ici

la classe SnakeWindow hérite de QFrame, qui permet de gérer une fenêtre simple. Comme tous les objets graphiques, QFrame hérite elle-même de QWidget.

2.3 Evénements

Les widgets répondent aux événements, déclenchés généralement par les actions de l'utilisateur. Certaines actions sont gérées en redéfinissant des méthodes virtuelles de QWidget et de ses sous-classes (méthodes de gestion d'événements), d'autres le seront par le biais des signaux et des slots. Voici un aperçu des méthodes de gestion d'événements les plus courantes :

- paintEvent() est appelée chaque fois que l'affichage du widget doit être rafraichi. Le rafraichissement peut être déclenché en appelant update() sur le widget. Chaque widget affichant un contenu personnalisé doit l'implémenter. Les opérations de desssin se font via un objet QPainter et ne peuvent avoir lieu que dans une méthode paintEvent() ou une fonction appelée par paintEvent().
- resizeEvent() est appelée lorsque le widget a été redimensionné.
- mousePressEvent() est appelée lors d'un appui sur un bouton de la souris, alors que le curseur de la souris est à l'intérieur du widget.
- mouseReleaseEvent() est appelée lorsqu'un bouton de la souris est relâché, alors que le curseur de la souris est à l'intérieur du widget.
- mouseDoubleClickEvent() est appelée lorsque l'utilisateur double-clique dans le widget.
- mouseMoveEvent() est appelée chaque fois que la souris se déplace pendant qu'un bouton de la souris est enfoncé. Cela peut être utile lors des opérations de glisser-déposer.
- keyPressEvent() est appelé lorsque l'utilisateur appuie sur une touche, pendant que le widget a le focus
- keyReleaseEvent() est appelée à chaque fois qu'une touche est relâchée, pendant que le widget a le focus
- wheelEvent() est appelée chaque fois que l'utilisateur tourne la molette de la souris pendant que le widget a le focus.

Toutes ces méthodes prennent en paramètre un pointeur sur une sous-classe QEvent. Dans le projet fourni, vous observez que la classe SnakeWindow redéfinit les méthodes paintEvent(), pour afficher le jeu, et keyPressEvent() pour effectuer les déplacements en fonction de l'appui sur certaines touches.

2.4 Signaux et slots

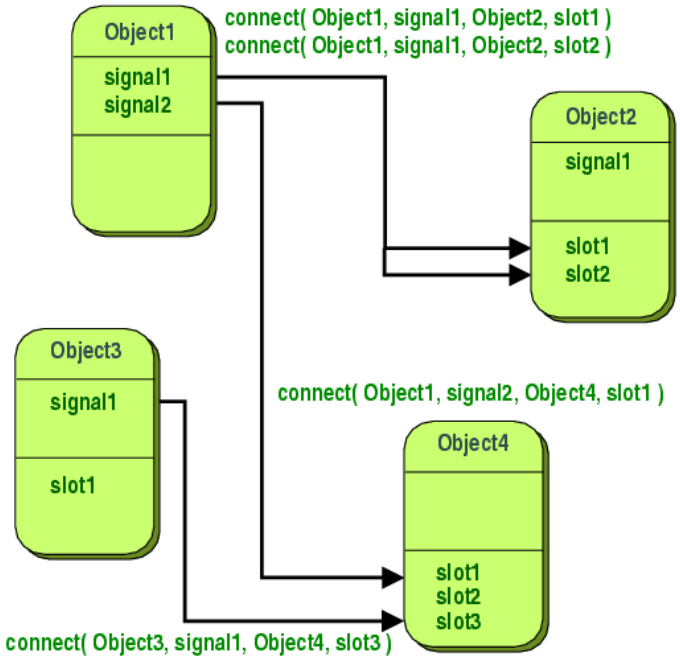
Les signaux et les slots sont utilisés pour la communication entre objets. Lorsqu'une action est effectuée sur un widget, on souhaite souvent qu'un autre widget soit notifié. Par exemple, si un utilisateur clique sur un bouton Fermer, on souhaite que la fonction close() de la fenêtre contenant ce bouton soit appelée. Un signal est un événement déclencheur, tandis que le slot est la fonction devant être appelée pour répondre à cet événement. Les widgets de Qt ont de nombreux signaux et slots prédéfinis, mais il est possible d'hériter des widgets pour ajouter ses propres signaux et slots.

Les signaux et les slots sont mis en place via les callbacks. Un callback est un pointeur sur une fonction. Un signal donné est connecté à un slot donné via un appel à la méthode connect (c'est une méthode statique de la classe QObject). Cette méthode existe en plusieurs versions. Parmi celles-ci, voici les deux prototypes qu'il vous est conseillé d'utiliser :

```
connect(objet1, signal1, objet2, slot2)
connect(objet1, signal1, slot)
```

La première version connecte le signal signal1 de l'objet objet1 au slot slot2 de l'objet objet2. On l'utilise lorsqu'on souhaite qu'une méthode (de la classe d'objet2) soit appelée lorsque survient le signal.

La deuxième version connecte le signal `signal1` de l'objet `objet1` à la fonction `slot` (c'est une fonction non-membre, pas une méthode).



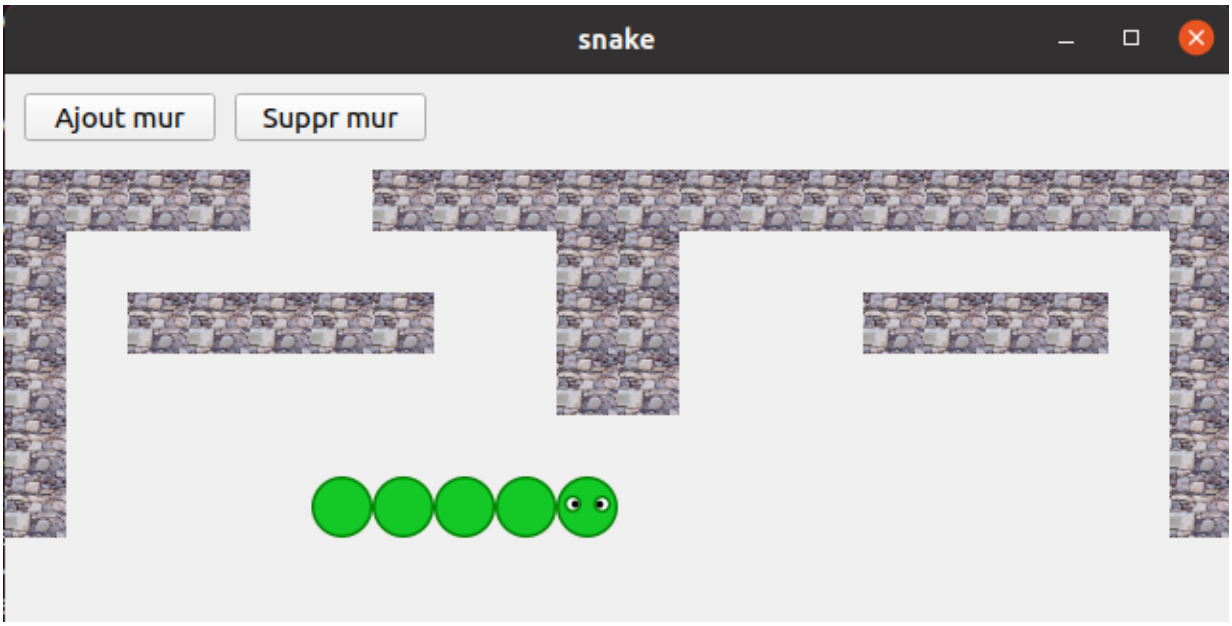
Un même signal peut être connecté à différents slots. Dans l'exemple précédent, si le signal `signal1` survient sur l'objet `Object1`, cela déclenchera l'appel des méthodes `slot1` et `slot2` sur l'objet `Object2`.

Dans le constructeur de `SnakeWindow`, vous observez qu'un objet de classe `QTimer` est créé, et que son signal `timeout` est connecté au slot `handleTimer` de l'objet courant. Notez que `timeout` est un signal prédéfini, tandis que `handleTimer` est une nouvelle méthode. Cette connexion fait en sorte qu'à chaque déclenchement du timer (toutes les 100ms), la méthode `handleTimer` sera appelée. C'est ce qui permet l'évolution du jeu et le rafraichissement de l'affichage.

3 Exercice

3.1 Ajout de boutons

Ajoutez deux boutons dans la fenêtre : un qui placera aléatoirement un mur, et l'autre qui en retirera un aléatoirement. Les boutons devront être positionnés en haut de la fenêtre. L'affichage du terrain devra donc être décalé vers le bas, comme le montre la figure suivante.



Les boutons sont gérés par la classe `QPushButton`. Ils peuvent être créés dans le constructeur de `SnakeWindow`, comme indiqué dans le modèle suivant :

```
SnakeWindow::SnakeWindow(QWidget *pParent, Qt::WindowFlags flags)
    :QFrame(pParent, flags)
{
   ...
   QPushButton *btn = new QPushButton(this);
   ...
   connect(btn, &QPushButton::clicked, this, &SnakeWindow::clickMonBouton);
}
```

Dans cet exemple, un bouton est créé avec, pour widget parent, la fenêtre courante. Le signal prédéfini `clicked` de la classe `QPushButton` est connecté à notre méthode `clickMonBouton` de `SnakeWindow`. Avant leur connexion, il vous faudra définir le texte, les dimensions et la position des boutons. Cherchez dans la documentation officielle de Qt.

A partir du moment où vous aurez ajouté un bouton, vous observerez que l'appui sur les touches flèches n'a plus d'effet sur le déplacement du serpent. Ceci est dû au fait que le bouton a le focus, et que c'est désormais lui qui capture les événements liés au clavier. La méthode `keyPressEvent` de `SnakeWindow` n'est donc plus appelée.

Pour y remédier, créez une classe fille de `QPushButton` et redéfinissez sa méthode `keyPressEvent`, de manière à ce qu'elle passe les événements clavier à sa fenêtre parent :

```
void SnakeButton::keyPressEvent(QKeyEvent *e)
{
    if (parent()!=nullptr)
        QCoreApplication::sendEvent(parent(), e);
}
```

Remplacez les instances de `QPushButton` par des instances de cette nouvelle classe. Si sa méthode `keyPressEvent` est correcte, l'objet de classe `SnakeWindow` doit à nouveau recevoir les événements liés au clavier.

3.2 Projet

Le projet sera réalisé en binôme (au sein du même groupe TP). Il consistera à ajouter une ou plusieurs fonctionnalité(s) de votre choix au jeu actuel. Voici des exemples :

- apparition aléatoire de fruits. Le serpent s'allonge quand il en mange un
- collision avec les murs
- gestion de la fin du niveau : porte de sortie,
- plusieurs serpents
- plusieurs niveaux
- changement de jeu (sur le même modèle : vue de dessus, en case par case) : Bomberman, Tron, Pacman, ...

Le projet devra être rendu par mail à votre intervenant de TP vendredi 12 avril à 23h59 au plus tard.

Le barème utilisé pour la notation du projet vous est donné à titre indicatif :

- **Rendu**
  - Consignes de rendu respectées : le projet est rendu en temps et en heure, aucun fichier ne manque, aucun fichier en trop (exe, fichiers .o, bibliothèques) : entre -2 et 1 pt
- **Technique**
  - Impression d'ensemble, Qualité de la réalisation, Difficulté technique apparente : 4 pts
  - Compilation et exécution : le projet compile sans erreur. Le programme s'exécute sans planter : 2 pts
  - L'utilisation du programme est documentée (soit dans les commentaires, soit dans un document à part) : 2 pts
- **Conception**
  - Ajout de classes, héritage des classes d'objets graphiques de Qt : 2 pts
  - Respect des principes de programmation objet (par exemple, les attributs/méthodes à ne pas exposer sont privés ou protégés), modélisation cohérente : 4 pts
  - Qualité d'écriture du code : indentation, nommage des variables, utilisation des boucles plutôt que du copier-coller, commentaires : 3 pts
  - Utilisation de la STL : string, vector, list, ofstream/ifstream : 2 pts

Consignes diverses :

- Afin de limiter la taille de l'archive rendue (maximum 1 Mo voire 2Mo) :
  - Ne mettez pas les bibliothèques Qt et FMOD. Supprimer les dossiers **bin** et **obj** générés par Code::Blocks. Votre archive ne devra contenir que les fichiers sources, le Makefile ou le fichier de projet Code::Blocks, les images et les sons éventuels.
  - Si vous ajoutez du son à votre projet, en utilisant FMOD, n'utilisez le format .wav que pour les sons très courts (comme dans l'exemple **BeatBox** disponible sur Celene). Pour les morceaux complets, utilisez le format .mp3
- Afin que votre code soit compilable sur n'importe quelle plateforme :
  - N'utilisez que des fonctions déclarées dans des en-têtes standards, Qt et FMOD (par exemple, l'utilisation de <windows.h> est à proscrire)
  - Prévoyez que votre programme puisse s'exécuter sur un système d'exploitation avec un système de fichier sensible à la casse (minuscules/majuscules), ce qui n'est pas le cas de Windows. Par exemple, si une image dans le dossier **data** s'appelle **SnAke.bMp**, la chaîne passée en argument à `QPixmap::load` doit être rigoureusement `"./data/SnAke.bMp"`.