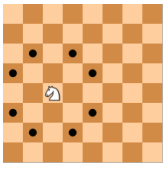
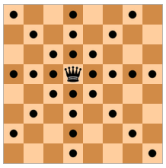
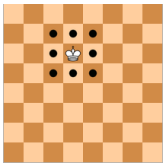
1. Jeu d'échecs (1)

Héritage, Classe abstraite, Redéfinition de méthode, Opérateurs d'affectation et de comparaison, Opérateur de sortie.

Le programme qu'il est demandé d'écrire permettra de simuler une partie d'échecs. La simulation ne sera que partielle car seules certaines règles du jeu seront prises en compte.

1. Une partie d'échecs se joue sur un échiquier, qui peut être vu comme une grille de 8 colonnes sur 8 lignes = 64 cases. Sur chacune de ces cases pourra être placée une pièce. Écrire une classe `position` permettant de stocker une position sur un échiquier. Il pourra être pertinent de définir un type `coordonnee` équivalant à un type entier signé, une position étant composée de deux `coordonnee`.
   `position` sera munie d'un constructeur à deux `coordonnee`, pas de constructeur par défaut, devra pouvoir être copiée (pour créer un nouvel objet par recopie d'un objet existant ou par un appel à l'opérateur d'affectation), disposer d'accesseurs et mutateurs, devra pouvoir être comparée par `==` et `!=` avec une autre `position`, et pouvoir être envoyée sur un flux de sortie.
   Définir une méthode `estvalide` retournant `true` si la position est valide sur un échiquier, `false` sinon.
2. Une pièce d'un échiquier dispose d'une couleur (parmi deux couleurs possibles : blanc et noir, définir un type énuméré pour cela) et d'une position. Une pièce peut être de différents types : pion, cavalier, dame, roi. À chacun de ces types correspond un ensemble de déplacements possibles (par exemple, le pion peut avancer d'une case, alors que la dame peut avancer de plusieurs cases à chaque déplacement), un symbole (une lettre utilisée pour l'affichage) et une valeur, qui est un entier représentant l'importance de la pièce. Noter que la valeur, le symbole et les déplacements dépendent uniquement du type de la pièce (tous les pions ont la même valeur, tous les pions peuvent se déplacer de la même façon, etc.). Les valeurs, symboles et déplacements sont décrits dans le tableau ci-dessous.
   Définir une classe `piece` ainsi que des sous-classes de `piece` pour chacun des 4 types de pièces.
   Munir ces classes de possibilités d'accéder à la valeur, au symbole, et à la position actuelle de la pièce.
   Écrire une méthode `tostring` qui retournera une chaîne du type `tcxy`, où `t` est le symbole du type, `c` la couleur (B ou N), `x` la colonne, `y` la ligne. Par exemple `CN42` représentera le cavalier noir en 4,2.

Pièce	Valeur	Symbole	Déplacements possibles
Pion	1	P	Pions blancs : une case vers le haut. Pions noirs : une case vers le bas. Lors du premier déplacement, une ou deux cases dans la direction correspondant à la couleur. Dans un premier temps, vous ne prendrez pas en compte cette particularité du premier déplacement d'un pion.
Cavalier	3	C	Une case dans une direction, deux cases dans une direction perpendiculaire à la première direction. 
Dame	9	D	Déplacement en diagonale, sur les colonnes ou sur les lignes. 
Roi	20	R	Déplacement d'une seule case en diagonale, sur les colonnes ou sur les lignes. 

3. Munir ces classes d'une méthode `deplacementspossibles` retournant l'ensemble des positions accessibles pour la pièce, cela dépend évidemment du type de pièce et de la position courante de la

pièce.
Définir la méthode `accepterposition` prenant comme paramètre une `position` et retournant un booléen valant `true` si la position passée en paramètre peut être atteinte en fonction de la position actuelle de la pièce, `false` sinon.
Définir la méthode `deplacer` prenant comme paramètre une position `destination` et déplaçant la pièce à la position `destination` (si la position est acceptable) et retournant `true` si le déplacement a été effectué, `false` s'il est impossible.

2. Bibliothèque

Héritage, Classe abstraite, Redéfinition de méthode, Polymorphisme, Opérateur de sortie, Constructeur virtuel.

Une bibliothèque offre à ses usagers la possibilité d'emprunter des documents. Selon la nature de ces documents, les conditions de prêt sont différentes. La bibliothèque dispose de vidéos, de livres et de périodiques. Les périodiques ne peuvent pas être empruntés. Les vidéos peuvent être empruntées. Certains livres peuvent être empruntés, mais d'autres non. Les informations à représenter concernent le titre et le nom de l'auteur. En plus, pour les périodiques, il est demandé de stocker le numéro. Pour les vidéos, il est demandé de représenter le type de support (DVD, Blu-Ray, Blu-Ray3D). Enfin, le nombre de pages doit être représenté pour les documents écrits, car il permet de calculer le coût du document (ce coût est imputé à la personne ayant emprunté le document en cas de perte, ou à la personne consultant le document en cas de détérioration) : Le coût des documents écrits est de 0,50 € la page, alors que le coût d'une vidéo est le même pour toutes les vidéos : 70 €.

1. Définir des classes pour représenter les différents types de documents, la possibilité de les emprunter, et leur coût, en factorisant au mieux le code. Tester les classes en écrivant une fonction `main` appelant les différentes méthodes afin de tester leur fonctionnement.
2. Écrire un opérateur de sortie sur flux fonctionnant sur les différents types de document et affichant le *Type (livre, périodique, vidéo), Titre, Nom de l'auteur, Empruntable (ou non empruntable), Coût*, suivi des informations supplémentaires spécifiques à certains types de documents : *nombre de pages, type de support vidéo, numéro du périodique*.
3. Écrire une classe `bibliotheque` contenant un ensemble de documents. On ne déclarera dans cette classe qu'un seul conteneur pour manipuler tous les documents. Dans cet exercice, on utilisera les pointeurs bruts, les `std::unique_ptr` et `std::shared_ptr` seront utilisés dans les exercices suivants.
4. Écrire une (des) méthode(s) permettant d'ajouter un document à la `bibliotheque`.
5. Écrire une (des) méthode(s) permettant d'accéder aux documents depuis l'extérieur de la classe, par exemple pour les parcourir afin de les afficher par l'opérateur de sortie sur flux, mais sans pouvoir les modifier, et sans pouvoir « casser » la classe `bibliotheque`.
6. Écrire une méthode prenant comme paramètre un type de support vidéo et retournant le nombre de vidéos de la bibliothèque qui sont de ce type.
7. Écrire une méthode permettant de faire une copie d'une bibliothèque. Attention : la copie de la bibliothèque contiendra des copies des documents de la bibliothèque originale (et non des pointeurs sur les mêmes instances). Le code de cette méthode devra gérer d'autres types de documents (qui n'ont pas encore été définis) sans aucune modification.

3. 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++ 14. 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 `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.

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).