1. Tableau à intervalle d'indices

Exceptions, std::vector, Surcharge d'opérateurs, Types emboîtés (chapitre 7).

Les éléments des tableaux C ou C++ sont repérés par des indices qui sont des entiers positifs commençant à 0. On veut ici écrire une classe permettant d'utiliser des tableaux dans lesquels l'intervalle des indices est quelconque, par exemple [-10,10] ou [10000,10100].

1. Déclarer une classe `arrayint` pour manipuler des tableaux d'entiers à indices quelconques. Vous utiliserez un `std::vector` pour représenter les éléments d'un `arrayint`.
   Munir cette classe d'un constructeur permettant de choisir l'indice minimum et l'indice maximum du tableau, ces indices étant des entiers signés.
   Définir deux accesseurs permettant d'accéder à l'indice minimum et à l'indice maximum du tableau.
   Définir un constructeur par recopie.
   Une déclaration d'un arrayint pourra ainsi se faire sous cette forme :
   ```
   arrayint tab(-10,10);
   arrayint tab2(tab);
   ```
   Les deux `arrayint` ainsi déclarés sont composés de 21 cases repérées par des indices compris entre -10 et 10.
2. Définir une classe `exceptionarrayint`, sous-classe de `std::exception`. Cette classe représentera les accès incorrects à un `arrayint`, c'est-à-dire l'accès à une valeur d'indice qui n'est pas incluse dans l'intervalle d'indices. Une instance d'`exceptionarrayint` mémorisera un message d'erreur (`std::string`) et l'indice erroné. Définir un constructeur permettant d'initialiser ces deux attributs. Définir la méthode `what`.
3. Écrire une méthode `at` dans `arrayint` prenant comme paramètre un indice et retournant l'entier situé à cet indice dans le tableau. Si l'indice passé ne fait pas partie de l'intervalle d'indices, lever une `exceptionarrayint`. Écrire l'opérateur d'accès [] faisant la même chose que `at` et permettant d'accéder au contenu d'un `arrayint` de la même façon qu'à un tableau C/C++. Par exemple sur l'`arrayint` déclaré ci-dessus, on pourra écrire : `std::cout << tab[-10];`.
4. Écrire une méthode `set` prenant comme paramètre un indice et une valeur et modifiant la valeur mémorisée dans le tableau pour l'indice passé.
5. Faire en sorte qu'il soit possible de faire une copie par affectation (=) et une comparaison d'égalité (==) sur des `arrayint`.
6. Écrire l'opérateur de sortie sur un flux <<. À titre d'exercice, cet opérateur utilisera l'accesseur à l'indice minimum du tableau, mais n'utilisera pas l'accesseur à l'indice maximum (et ne sera pas `friend` d'`arrayint`, et on ne rajoutera pas de méthode retournant la taille d'un `arrayint`).
7. Définir un type emboîté `const_iterator` dans `arrayint` et munir `arrayint` de méthodes `begin` et `end` retournant un `const_iterator` afin de pouvoir parcourir un `arrayint` de la même façon qu'un conteneur de la bibliothèque standard : (soit `tab` un `arrayint`)
   ```
   for (arrayint::const_iterator i=tab.begin(); i!=tab.end(); ++i)
     std::cout << (*i) << "\n";
   ```
8. (question optionnelle) Écrire une sous-classe d'`arrayint` qui propose les mêmes fonctions, mais avec une fonction supplémentaire : Les instances de cette sous-classe mémoriseront à tout moment dans quelles cases des valeurs ont déjà été stockées, les autres cases contenant des « valeurs non initialisées ». Lors de l'accès en lecture à une case non initialisée (par `at` ou `[]`), une exception sera levée, ce qui facilitera le débogage du programme (même si Valgrind est capable de faire la même chose, sans requérir la moindre modification du code source du programme). Attention au constructeur par recopie (et à l'opérateur d'affectation et de comparaison) : Copier un tableau non initialisé créera un tableau non initialisé, mais ne lèvera pas d'exception.

2. Résultats du loto, utilisation de `std::map`

std::list, std::map, Exceptions, Surcharge d'opérateurs, Algorithmes de la bibliothèque standard.

`std::map` est un conteneur associatif, permettant d'associer des valeurs à des clés. Les types correspondant aux clés et aux valeurs sont les paramètres du modèle de classes : par exemple `std::map<std::string, int>` est un conteneur dans lequel les clés sont des chaînes et les valeurs des entiers. Sur un tel conteneur `m`, on pourra écrire `m["abc"] = 3;` car l'opérateur `[]` permet d'accéder à la valeur associée à la clé. Un parcours (par itérateur ou par boucle `for` d'intervalle) d'un `std::map` permet d'accéder à des valeurs qui sont de type `std::pair<clé,valeur>`, qui est une `struct` disposant de deux champs `first` (clé) et `second` (valeur). Ainsi `for (auto i : m) std::cout << i.first;` affichera toutes les clés d'un `std::map m`.
Le type utilisé pour la clé devra posséder un opérateur de comparaison < afin que `std::map` puisse comparer les clés et insérer un nouvel élément à l'endroit convenu dans la structure de données interne (qui est un arbre binaire de recherche afin de fournir des traitements avec une bonne complexité : insertion et accès en temps logarithmique).
Plus d'informations : http://en.cppreference.com/w/cpp/container/map

1. Écrire une classe `resultat` qui mémorise le résultat d'un tirage du loto. Un `resultat` sera formé de 5 entiers compris entre 1 et 49. Les entiers seront stockés dans une liste d'entiers, qui sera un attribut de la classe. Définir un constructeur sans argument (construisant une `resultat` vide), et faire en sorte que les `resultat` soient copiables.
2. Définir une méthode `ajouternumero` qui ajoute au `resultat` l'entier passé en paramètre. Cette méthode lèvera une exception dans le cas où l'entier passé en paramètre n'est pas compris entre 1 et 49 ou s'il figure déjà dans la liste des numéros du `resultat` (pour cela, vous n'écrirez pas une boucle de recherche mais utiliserez un algorithme de la bibliothèque standard http://en.cppreference.com/w/cpp/algorithm), ou si le `resultat` contient déjà 5 numéros. Vous utilisez une classe exception fournie par la bibliothèque standard afin d'éviter de définir une nouvelle classe : `std::invalid_argument` http://en.cppreference.com/w/cpp/error/invalid_argument
3. Écrire une méthode `trier` de `resultat` qui trie dans l'ordre croissant les numéros stockés dans la liste. Écrire un opérateur de sortie de `resultat` sur un flux.
4. Écrire une classe `date` formée de trois attributs entiers : année, mois, jour. Définir constructeur, accesseurs, aucun mutateur : la date doit être fixée à la construction, opérateur d'affectation =, opérateur de sortie sur un flux, opérateurs de comparaison ==, !=, <, <=, >, >=.
5. Définir une classe `ensembleresultat` permettant de mémoriser un ensemble de résultats avec la date du tirage, en ayant un accès efficace à résultat d'une date donnée.
   Écrire une méthode `ajout` prenant comme paramètre une `date` et un `resultat` et ajoutant à l'ensemble le résultat associé à la date passée. S'il y avait déjà un résultat à cette date, il est remplacé par celui passé en paramètre.
   Écrire une méthode qui en fonction d'une `date`, retourne le `resultat` correspondant à cette date (et lève une exception s'il n'y a pas de résultat mémorisé pour cette date).
   Écrire une méthode retournant l'ensemble trié des `dates` pour lesquelles un `resultat` est mémorisé.

3. Séquence de couleurs (5) : Interface graphique

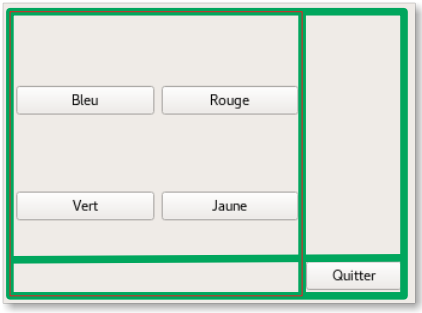
QGridLayout, Feuille de style Qt, QPushButton, QLineEdit, QLabel, QMessageBox, std::map.

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 ici :
   Faire en sorte qu'une `couleur` puisse être envoyée sur un flux de sortie et affiche bleu, jaune, etc.
   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 puisse être faite par un constructeur par

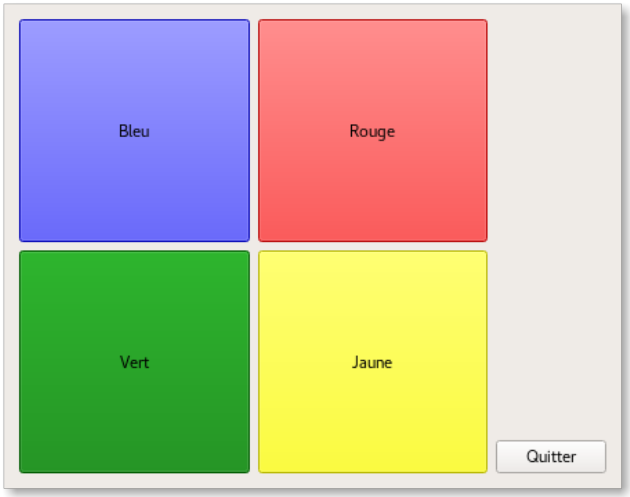
recopie et un opérateur d'affectation.
Supprimer la méthode `comparer` et faire en sorte que la comparaison se fasse par l'opérateur ==.
Supprimer la méthode `acces` et faire en sorte que l'accès à un élément de la séquence soit fait avec l'opérateur [].
Écrire une méthode `vider` qui vide la séquence.

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.

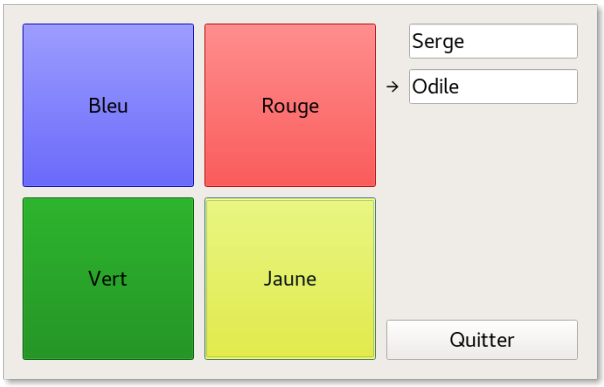
4. Déclarer un attribut de type `std::map` dans `simon` permettant, à partir d'une `couleur` (clé), d'accéder au `QPushButton*` (valeur) représentant la `couleur`, ce `map` étant évidemment rempli par le constructeur de `simon`.
Écrire une méthode privée `boutonverscouleur` qui à partir d'un `QPushButton*`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