Laboratoire de Programmation en C++

2ème bachelier en informatique :
orientations « informatique industrielle », « réseaux et télécommunications » et « développement d'applications »
(1er quadrimestre)
Année académique 2023-2024

« Mini-PhotoShop » en C++

Anne Léonard
Patrick Quettier
Jean-Marc Wagner

Introduction

1. Informations générales : UE, AA et règles d'évaluation

Cet énoncé de laboratoire concerne les unités d'enseignement (UE) suivantes :

a) 2ème Bach. « Développement d'applications » : « Développement Système et orienté objet »

Cette UE comporte les activités d'apprentissage (AA) suivantes :

- **Base de la programmation orientée objet – C++** (45h, Pond. 30/90)
- **Principes fondamentaux des Systèmes d'exploitation** (15h, Pond. 10/90)
- **Système d'exploitation et programmation système UNIX** (75h, Pond. 50/90)

Ce laboratoire intervient dans la construction de la côte de l'AA « Base de la programmation orientée objet – C++ ».

b) 2ème Bach. « informatique et systèmes » : « Développement Système et orienté objet »

Cette UE comporte les activités d'apprentissage (AA) suivantes :

- **Base de la programmation orientée objet – C++** (45h, Pond. 45/101)
- **Système d'exploitation et programmation système UNIX** (56h, Pond. 56/101)

Ce laboratoire intervient dans la construction de la côte de l'AA « Base de la programmation orientée objet – C++ ».

Quel que soit le bachelier, la cote de l'AA « Base de la programmation orientée objet – C++ » est construite de la même manière :

♦ théorie : un examen écrit en janvier 2024 et coté sur 20

♦ laboratoire **(cet énoncé)** : une évaluation globale en janvier fournissant une note de laboratoire sur 20. Des « check-points » réguliers formatifs auront lieu pendant tout le quadrimestre.

♦ note finale : **moyenne géométrique de la note de théorie (50%) et de la note de laboratoire (50%)**.

Cette procédure est d'application tant en 1ère qu'en 2ème session.

1) Chaque étudiant doit être capable d'expliquer et de justifier l'intégralité du travail.

2) En 2ème session, un **report de note** est possible pour la théorie ou le laboratoire **pour des notes supérieures ou égales à 10/20**. Les évaluations (théorie ou laboratoire) ayant des **notes inférieures à 10/20** sont **à représenter dans leur intégralité**.

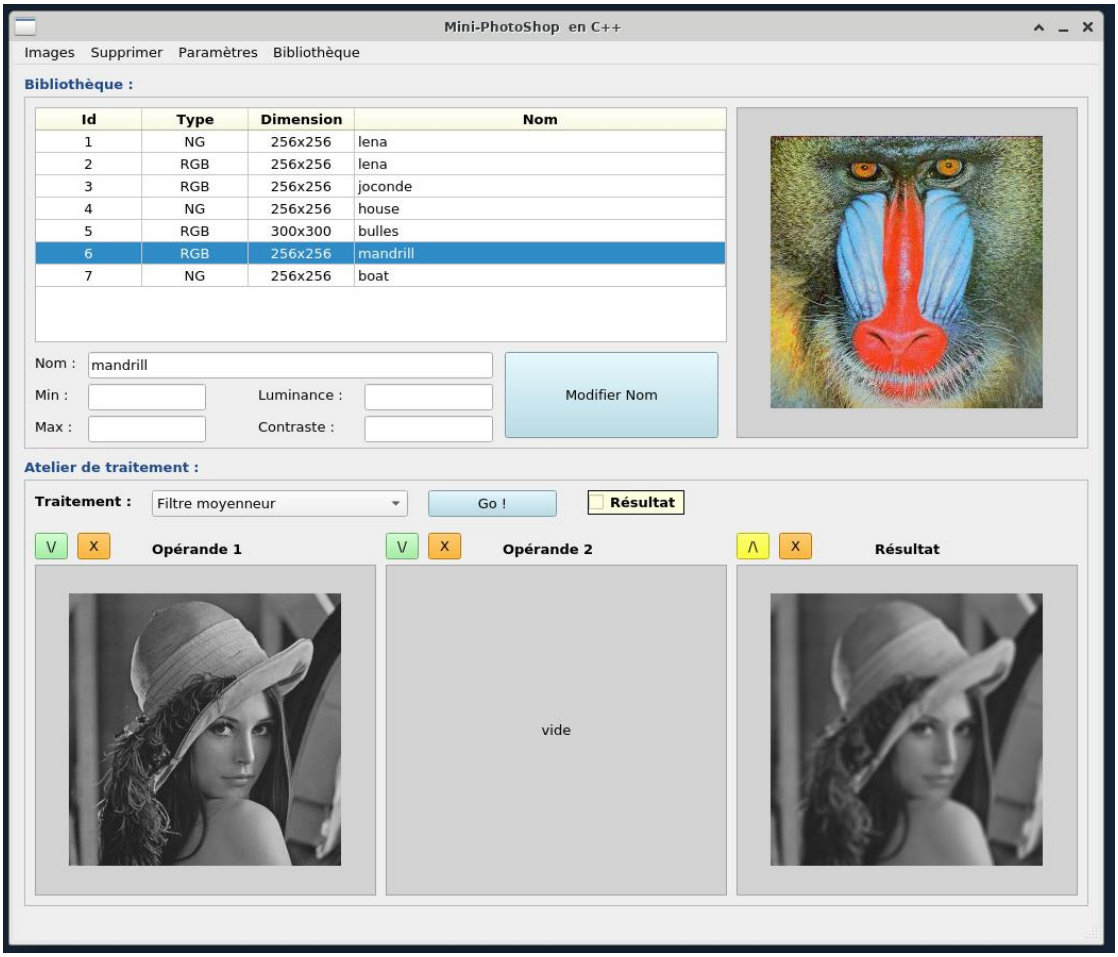
3) Les consignes de présentation des travaux de laboratoire sont fournies par les différents professeurs de laboratoire

2. Le contexte : Introduction au traitement d'images

Les travaux de Programmation Orientée Objets (POO) C++ consistent à réaliser une application introductive au traitement d'images. Tout d'abord, il faut savoir qu'il existe trois types d'images :

- les images couleurs, dites RGB (« Red », « Green », « blue ») où chaque pixel est associé à une couleur qui, elle-même, possède 3 composantes à valeur entière comprise entre 0 et 255.
- les images en niveaux de gris où chaque pixel est associé à une valeur entière comprise entre 0 (noir) et 255 (blanc). Il y a donc 256 niveaux de gris en tout.
- les images binaires où chaque pixel est associé à une valeur booléenne. La valeur « true » est généralement associée au blanc, et la valeur « false » au noir, mais ce n'est pas une obligation.

L'application aura l'aspect visuel suivant :



Il s'agit d'une interface graphique basée sur la librairie C++ Qt. Celle-ci vous sera fournie telle quelle. Vous devrez programmer la logique de l'application et non concevoir l'interface graphique.

Différentes entités vont donc apparaître : les notions d'images, de couleur,… Une fois que toutes ces entités auront été mises en place, différentes opérations élémentaires du traitement d'images seront abordées dans une application :

- la conversion entre images couleurs et images en niveaux de gris
- le floutage d'une image
- la réduction du « bruit » parasite
- la détection des objets et formes présents dans une image
- …

3. Philosophie du laboratoire

Le laboratoire de programmation C++ sous **Linux** a pour but de vous permettre de faire concrètement vos premiers pas en C++ au début du quadrimestre (septembre-octobre) puis de conforter vos acquis à la fin du quadrimestre (novembre-décembre). Les objectifs sont au nombre de trois :

- mettre en pratique les notions vues au cours de théorie afin de les assimiler complètement,
- créer des "briques de bases" pour les utiliser ensuite dans une application de synthèse,
- vous aider à préparer l'examen de théorie du mois de janvier.

Il s'agit bien d'un laboratoire de C++ sous Linux. Il a également pour but de vous familiariser à un environnement de développement autre que Windows mais surtout en dehors d'un IDE totalement fenêtré. Pour cela, on vous fournira une **machine virtuelle (VMWare)** dès le premier laboratoire.

4. Méthodologie de développement

La programmation orientée objet permet une approche modulaire de la programmation. En effet, il est possible de scinder la conception d'une application en 2 phases :

1. La programmation des classes de base de l'application (les briques élémentaires) qui rendent un service propre mais limité et souvent indépendant des autres classes. Ces modules doivent respecter les contraintes imposées par « un chef de projet » qui sait comment ces classes vont interagir entre elles. Cette partie est donc réalisée par « le programmeur créateur de classes ».
2. La programmation de l'application elle-même. Il s'agit d'utiliser les classes développées précédemment pour concevoir l'application finale. Cette partie est donc réalisée par « le programmeur utilisateur des classes ».

Durant la première partie de ce laboratoire (**de la mi-septembre à mi-novembre**), vous vous situez en tant que « programmeur créateur de classes ». On va donc vous fournir une série de jeux de tests (les fichiers **Test1.cpp**, **Test2.cpp**, **Test3.cpp**, …) qui contiennent une fonction main() et qui vous imposeront le comportement (l'interface) de vos classes. Cette méthodologie de développement est

bien connue en entreprise, il s'agit de la méthodologie appelée « **Test-Driven Development (TDD)** » :

Le **Test-Driven Development (TDD)**, ou développement piloté par les tests en français, est une méthode de développement de logiciel qui consiste à concevoir un logiciel par petites étapes, de façon progressive, en écrivant avant chaque partie du code source propre au logiciel les tests correspondants et en remaniant le code continuellement. (wikipedia)

Dans la deuxième partie du laboratoire (**de mi-novembre à fin décembre**), vous vous situerez en tant que « programmeur utilisateur des classes » utilisant les classes que vous aurez développées précédemment. C'est dans cette seconde phase que vous développerez l'application proprement dite.

5. Code source de base

Tous les fichiers sources de base fournis peuvent être obtenus par

- Clonage du repository **GitHub** : https://github.com/hepl-dsoo/LaboCpp2023_Enonce

6. Planning et contenu des évaluations

a) Evaluation 1 (formative → donc non certificative) :

Le développement de l'application, depuis la création des briques de base jusqu'à la réalisation de l'application avec ses fonctionnalités a été découpé en une **série d'étapes à réaliser dans l'ordre**. A chaque nouvelle étape, vous devez rendre compte de l'état d'avancement de votre projet à votre professeur de laboratoire qui validera (ou pas) l'étape.

b) Evaluation 2 (certificative → examen de janvier 2024) :

Porte sur :

- la validation des étapes non encore validées le jour de l'évaluation,
- le développement et les tests de l'application finale.
- Vous devez être capable d'expliquer l'entièreté de tout le code développé.

Date d'évaluation : jour de votre examen de Laboratoire de C++ (selon horaire d'examens)

Modalités d'évaluation : Sur la machine Linux fournie, selon les modalités fixées par le professeur de laboratoire.

Plan des étapes à réaliser

CONTRAINTES : Tout au long du laboratoire de C++ (évaluation 1 et 2, et seconde session), il vous est interdit, pour des raisons pédagogiques, d'utiliser les **containers génériques template de la STL**.

Etape 1 (Test1.cpp) : Une première classe

a) Description des fonctionnalités de la classe

Un des éléments principaux en traitement d'images est forcément la notion d'images. Dans un tout premier temps, une image aura tout d'abord un nom ainsi qu'un identifiant numérique entier. Notre première classe, la classe **ImageNG** (NG = niveaux de gris), sera donc caractérisée par :

- **id** : un entier (**int**) représentant l'identifiant de l'image. Celui-ci permettra à terme d'identifier une image de manière unique dans l'application.
- **nom** : une chaine de caractères allouée dynamiquement (**char ***) en fonction du texte qui lui est associé. Il s'agit du nom de l'image.

Comme vous l'impose le premier jeu de test (Test1.cpp), on souhaite disposer au minimum des trois formes classiques de constructeurs et d'un destructeur, des méthodes classiques getXXX() et setXXX() et une méthode pour afficher les caractéristiques de l'objet. La variable **nom** de type chaîne de caractères sera actuellement un char*. **Pour des raisons purement pédagogiques, le type string (de la STL) NE pourra PAS être utilisé dans cette 1ère étape de ce dossier de C++.** Vous aurez l'occasion d'utiliser la classe string dans la suite de ce projet.

b) Méthodologie de développement

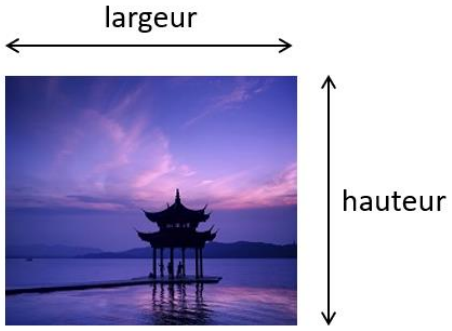
Veillez à tracer (cout << …) vos constructeurs et destructeurs pour que vous puissiez vous rendre compte de quelle méthode est appelée et quand elle est appelée.

On vous demande de créer pour la classe ImageNG (ainsi que pour chaque classe qui suivra) les **fichiers .cpp et .h** et donc de travailler en fichiers séparés. Un **makefile** permettra d'automatiser la compilation de votre classe et de l'application de tests.

Etape 2 (Test2.cpp) : Associations entre classes : agrégation + Variables statiques

a) La classe Dimension (+ variables membres statiques)

Une image possède ensuite des dimensions que nous appellerons largeur et hauteur :



et qui sont exprimées en nombre de pixels. Dès lors, nous allons compléter notre classe ImageNG. Mais avant cela, on vous demande de développer la classe **Dimension**, qui est caractérisée par :

- **largeur**, **hauteur** : deux entiers (**int**) strictement supérieurs à 0.

Différents constructeurs sont demandés, comme vous pourrez le voir dans le jeu de Test2.cpp. La dimension construite par le constructeur par défaut sera (largeur,hauteur) = (400,300).

Dans le monde de l'image et de la vidéo, il y a des dimensions d'image qui sont normalisées, comme VGA, HD ou Full HD. Dès lors, on pourrait imaginer de créer des **objets « permanents »** (dits **« statiques »**) existant même si le programmeur de l'application n'instancie aucun objet et représentant ces dimensions particulières. Dès lors, on vous demande d'ajouter, à la classe Dimension, 3 variables membres publiques, appelées VGA, HD et FULL_HD, statiques, constantes de type Dimension et ayant les valeurs respectives (640,480), (1280,720) et (1920,1080). Voir jeu de tests.

b) Modification de la classe ImageNG (agrégation par valeur)

Afin de compléter la classe **ImageNG**, on vous demande de lui ajouter une nouvelle variable membre appelée **dimension** dont le type est **Dimension (et non Dimension* !)**. De plus,

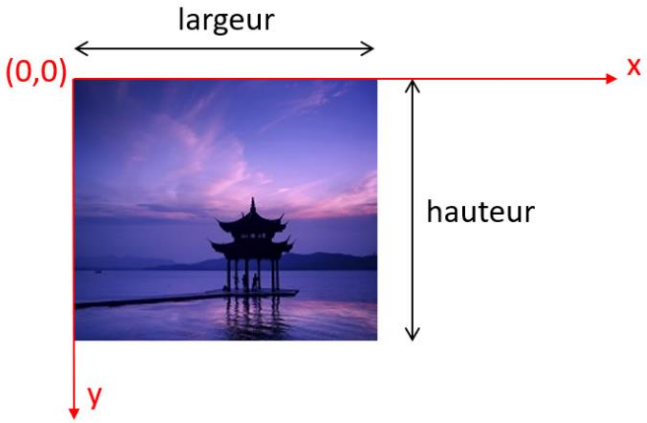
- Vous ne pouvez pas toucher aux prototypes des constructeurs existants ! Vous devez simplement vous arranger pour que les images créées par ces constructeurs aient la dimension par défaut.
- Vous devez ajouter un nouveau constructeur d'initialisation complet tenant compte de la dimension.
- N'oubliez pas les setter/getter correspondant à la nouvelle variable membre dimension et de mettre à jour la méthode Affiche() de ImageNG.

Etape 3 (Test3.cpp) : Mise en place de la matrice de pixels Utilisation de méthodes statiques, agrégation par utilisation

Comme déjà mentionné, dans un premier temps, nous nous contenterons de représenter une image en niveau de gris. Pour rappel, un niveau de gris est une valeur entière comprise entre 0 et 255.

a) Matrice de pixels

Les pixels seront stockés dans une matrice (tableau à 2 dimensions). Dans ce tableau, les **colonnes** et les **lignes** correspondront aux coordonnées **(x,y)** des pixels de l'image :



Pour des raisons de simplification du code, on fixera la taille maximale d'une image en allouant la matrice de pixels de manière statique :

```
int matrice[L_MAX][H_MAX] ;
```

où

- **matrice** est une variable membre à ajouter à la classe **ImageNB**,
- **L_MAX** et **H_MAX** sont des constantes entières (variables membres statiques constantes de la classe **ImageNG**) représentant respectivement la largeur maximale et la hauteur maximale qui peuvent être choisies égales à **500**. Les dimensions d'une image ne pourront donc pas dépasser 500x500 et on utilisera effectivement une sous-matrice.

Remarque (NON OBLIGATOIRE) : Pour les « puristes fans d'optimisation », la taille de la matrice pourrait être gérée de manière dynamique. Pour l'allocation d'une telle matrice, les normes actuelles du C++ vous imposent de coder

```
int** matrice = new int*[largeur];
for (int x=0 ; x<largeur ; x++) matrice[x] = new int[hauteur];
```

Vous pourrez ensuite utiliser la matrice sous sa forme classique **matrice[x][y].** Pour libérer l'espace mémoire d'une telle matrice, vous devrez coder

```
for (int x=0 ; x<largeur ; x++) delete[] matrice[x];
delete[] matrice;
```

Plusieurs consignes et remarques importantes :

- Il ne doit pas y avoir de méthodes **setMatrice** et **getMatrice** ! En effet, l'utilisateur (programmeur) de votre classe **ImageNG** n'a pas à savoir comment sont stockés les pixels dans l'image, ni à avoir un accès direct à cette matrice (afin d'éviter les problèmes). C'est le principe de l'encapsulation.

- C'est la variable **dimension** qui fixe effectivement la taille de l'image et donc de la sous-matrice utilisée dans la variable matrice.

- N'oubliez pas de mettre à jour les constructeurs de votre classe **ImageNG** tenant compte de cette nouvelle variable membre **matrice** !

- L'accès aux pixels devra se faire via deux méthodes d'instance de la classe **ImageNG**, méthodes que vous devez ajouter :
  - **void setPixel(int x,int y,int val)** où (x,y) sont les coordonnées du pixel à modifier et val la nouvelle valeur du niveau de gris.
  - **int getPixel(x,y)** qui retourne le niveau de gris du pixel dont on passe les coordonnées (x,y) en paramètres.

- On vous demande également de créer une méthode **void setBackground(int val)** permettant de mettre tous les pixels de l'image au même niveau de gris val.

Pour pouvoir tester tout cela, il serait intéressant de pouvoir visualiser l'image dans une fenêtre du système d'exploitation… Bin allons-y 😊 !

b) Visualisation d'une image : agrégation par utilisation

Afin de gérer l'aspect graphique des images, on vous donne une petite librairie fournie sous la forme d'une classe appelée **MyQT** (basée sur la librairie C++ Qt).

MyQT contient les méthodes statiques suivantes :

- **void ViewImage(const ImageNG & image)** : qui permet d'afficher une boîte de dialogue affichant l'image passée en paramètre (à condition que votre classe **ImageNG** respecte les conventions de nommage des méthodes imposées !). Une fois affichée, il faudra cliquer sur la croix de la fenêtre pour que le programme continue son exécution.
- **void importFromFile(ImageNG & image, const char* fichier)** qui permet de remplir l'objet image à partir d'un fichier au format courant comme BMP, PNG ou JPEG.
- **void exportToFile(const ImageNG & image, const char* fichier, const char* format)** qui permet de créer un fichier au format BMP, PNG, … à partir de l'objet image passé en paramètre. La chaîne de caractères format doit être « BMP », « PNG » ou « JPEG ».

Remarque importante : Pour compiler la classe **MyQT**, voici la ligne de compilation nécessaire :

```
g++ -c MyQT.cpp -pipe -O2 -std=gnu++11 -Wall -W -D_REENTRANT -fPIC -
DQT_DEPRECATED_WARNINGS -DQT_NO_DEBUG -DQT_WIDGETS_LIB -DQT_GUI_LIB -
DQT_CORE_LIB -I. -isystem /usr/include/qt5 -isystem
/usr/include/qt5/QtWidgets -isystem /usr/include/qt5/QtGui -isystem
/usr/include/qt5/QtCore -I. -I. -I/usr/lib64/qt5/mkspecs/linux-g++
```

et pour compiler le programme **Test3.cpp** et faire l'édition de lien :

```
g++ Test3.cpp -o Test3 MyQT.o ImageNG.o Dimension.o -lSDL -lpthread -Wl,-
O1 -pipe -O2 -std=gnu++11 -Wall -W -D_REENTRANT -fPIC -
DQT_DEPRECATED_WARNINGS -DQT_NO_DEBUG -DQT_WIDGETS_LIB -DQT_GUI_LIB -
DQT_CORE_LIB -I. -isystem /usr/include/qt5 -isystem
/usr/include/qt5/QtWidgets -isystem /usr/include/qt5/QtGui -isystem
/usr/include/qt5/QtCore -I. -I. -I/usr/lib64/qt5/mkspecs/linux-g++
/usr/lib64/libQt5Widgets.so /usr/lib64/libQt5Gui.so
/usr/lib64/libQt5Core.so /usr/lib64/libGL.so
```

Vous devez donc mettre à jour votre makefile en conséquence.

Nous allons à présent modifier la classe **ImageNG** afin qu'elle puisse se dessiner dans une fenêtre graphique. On vous demande d'ajouter à la classe **ImageNG** la méthode **Dessine()** qui appelle la méthode **ViewImage()** de la classe **MyQT** en lui passant ***this** en paramètre.

Remarque théorique : La méthode **Dessine()** de la classe **ImageNG** utilise l'**agrégation par utilisation**. En effet, elle utilise la classe **MyQT** (pour pouvoir dessiner dans la fenêtre graphique).

c) Importation/Exportation de et vers des fichiers

Nous allons à présent modifier la classe **ImageNG** afin qu'elle puisse importer/exporter une image à partir d'un/vers un fichier au format adéquat. On vous demande d'ajouter à la classe **ImageNG**

- la méthode **void importFromFile(const char* fichier)** qui appelle la méthode **ImportFormFile()** de la classe **MyQT** en lui passant ***this** en paramètre.
- la méthode **void exportToFile(const char* fichier, const char* format)** qui appelle la méthode **ExportToFile()** de la classe **MyQT** en lui passant ***this** et le format d'image en paramètres.

Etape 4 (Test4.cpp) : Surcharges des opérateurs

Il s'agit ici de surcharger un certain nombre d'opérateurs des classes développées ci-dessus afin d'en étendre les fonctionnalités.

a) Les paramètres d'une image (Essai1())

En traitement d'images, il est courant de calculer quelques paramètres importants d'une image comme sa luminance, son contraste, son minimum et son maximum. On vous demande avant toute chose d'ajouter les méthodes suivantes à votre classe **ImageNG** :

- **int getLuminance()** : qui retourne la luminance de l'image, c'est-à-dire la valeur moyenne des niveaux de gris de tous les pixels de l'image. Ce paramètre nous apporte de l'information sur le fait qu'une image est plutôt « claire » ou plutôt « sombre ».
- **int getMinimum()** et **int getMaximum()** : qui retournent respectivement le minimum et le maximum des niveaux de gris de tous les pixels de l'image. Ces paramètres constituent ce que l'on appelle la dynamique de l'image.
- **float getContraste()** : qui retourne le contraste de l'image qui se calcule par la formule

$$contraste = \frac{max - min}{max + min}$$

où max et min sont les maximum et minimum de l'image. La valeur obtenue est un nombre réel compris entre 0 et 1. Il est d'autant plus grand que visuellement les « objets » de l'image « ressortent » par rapport au fond de l'image.

b) Surcharge de l'opérateur = de la classe ImageNG (Essai2())

Vous devez créer l'opérateur d'affectation de la classe **ImageNG** afin que l'exécution de ce genre de code soit possible :

```
ImageNG i1(1,"essai",Dimension(400,300)),i,i2,i3 ;
i = i1 ;
i2 = i3 = ImageNG("essai.bmp"); // nouveau constructeur à faire !!!
i3.Affiche();
```

c) Surcharge des opérateurs << et >> de Dimension (Essai3()) et << de ImageNG (Essai4())

Vous devez ensuite créer les opérateurs << et >> de la classe **Dimension** et l'opérateur << de la classe **ImageNG** afin que l'exécution de ce genre de code soit possible :

```
Dimension d;
cin >> d ;
cout << d << endl ;

ImageNG i1("boat.bmp");
cout << "Votre image : " << i1 << endl ;
```

L'opérateur << de **ImageNG** devra afficher sur une seule ligne l'identifiant, le nom, la largeur, la hauteur, la luminance et le contraste de l'image.

d) Opérateurs (ImageNG ± int) et (int + ImageNG) de la classe ImageNG (Essai5(), Essai6()))

Additionner (soustraire) une valeur entière à une image va avoir pour effet d'ajouter (soustraire) cette valeur aux niveaux de gris de tous les pixels de l'image. Ces opérateurs vont donc avoir pour effet de **créer une nouvelle** image plus claire (sombre) que l'image de départ, **laissant cette dernière inchangée**. **Attention !** Les niveaux de gris devront rester dans l'intervalle [0,255] : tout niveau de gris dépassant 255 devra être ramené à 255 et tout niveau de gris passant sous 0 devra être ramené à 0.

```
ImageNG i1("boat.bmp"),i,i2,i3 ;
i = i1 + 70 ;
i = 20 + i ;
i2 = i1 - 40 ;
```

e) Surcharge des opérateurs ++ et -- de la classe ImageNG (Essai17() et Essai8())

On vous demande de programmer les opérateurs de post et pré-in(dé)crémentation de la classe **ImageNG**. Ceux-ci in(dé)crémenteront une **ImageNG** de **20**. Cela permettra d'exécuter le code suivant :

```
ImageNG i("boat.bmp");
cout << ++i << endl ;
cout << i++ << endl ;
cout << --i << endl ;
cout << i-- << endl ;
```

f) Surcharge des opérateurs de comparaison (== et !=) de la classe Dimension (Essai9())

Les dimensions de deux images sont soit identiques, soit différentes. On peut difficilement concevoir de les comparer. On vous demande donc simplement de surcharger les opérateurs == et != de la classe **Dimension** permettant d'exécuter un code du genre :

```
Dimension d1, d2;
…
if (d1 == d2) …
if (d1 != d2) …
```

Evidemment, deux dimensions sont égales si elles ont même largeur et même hauteur.

g) Surcharge de l'opérateur (ImageNG - ImageNG) de la classe ImageNG (Essai10())

Dans certains traitements d'image, il est intéressant de réaliser la différence entre deux images (exemple : détection des contours des objets). On vous demande donc de surcharger l'opérateur - de la classe **ImageNG** permettant d'exécuter un code du genre :

```
ImageNG i1("boat.bmp"),i2("boat2.bmp"),i;
i = i1 - i2 ;
```

Le résultat est une image et la différence se fait pixel à pixel. Si le résultat de la soustraction passe sous 0, il faudra ramener cette valeur à 0.

h) Surcharge des opérateurs de comparaison (< > et ==) de la classe ImageNG (Essai13())

Comparer deux images se fait pixel à pixel : pour qu'une image soit < (> ou ==) à une autre image, il faut que tous ses pixels soient < (> ou ==) à chaque pixel correspondant de l'autre image.

On vous demande de programmer les opérateurs <, > et == de la classe **ImageNG** qui comparent deux images :

```
ImageNG i1, i2;
…
if (i1 < i2) …
if (i1 > i2) …
if (i1 == i2) …
```

Etape 5 (Test5.cpp) : Associations de classes : héritage et virtualité

Il s'agit ici de mettre en place les classes nécessaires à la gestion des images « couleur » et « binaire ». On se rend vite compte que toutes ces images (avec celle en niveaux de gris) ont des caractéristiques communes comme l'**identifiant**, le **nom** et la **dimension**. Nous allons donc tout d'abord modifier la classe **ImageNG** afin qu'elle hérite d'une **classe abstraite** **Image** reprenant les caractéristiques communes de toutes les images, et cela sans que l'utilisation de la classe ImageNG ne soit modifiée par rapport à ce qui a déjà été fait (en d'autres termes les Test1.cpp, Test2.cpp, Test3.cpp, Test4.cpp devront continuer à compiler et fonctionner exactement comme auparavant – on parle de « refactoring » de code). Ensuite, on vous demande de développer la petite hiérarchie de classes décrite ci-dessous.

a) La classe abstraite Image – refactoring de la classe ImageNG

On vous demande de créer la classe abstraite **Image** qui possède

- les variables membres **id** (**int**), **nom** (**char* → string** si vous le souhaitez) et **dimension** (**Dimension**)
- les accesseurs getXXX/setXXX pour ces trois variables membres.
- Les **méthodes virtuelles pures** void **Affiche**(), void **Dessine**() et void **exportToFile**(const char* fichier, const char* format)

Etant donné que cette classe est **abstraite**, il est impossible de l'instancier et donc de réaliser des tests actuellement. Elle sert simplement à donner aux futures classes héritées :

- du code commun et un service : l'id, le nom et la dimension
- une interface commune : toutes les classes héritées auront les méthodes Affiche, Dessine et exportToFile

Ensuite, vous devez modifier la classe **ImageNG** de telle sorte qu'elle hérite de la classe **Image** mais dispose en plus de

- une variable membre **matrice** de **int** représentant la matrice de pixels (valeurs de niveaux de gris)
- toutes les méthodes et opérateurs dont disposait déjà la classe **ImageNG**.

En fait, la seule chose qui change est que la gestion des variables membres communes à toutes les classes d'image ne sont plus gérées par la classe **ImageNG** mais par sa classe mère **Image**.

Pour tester les classes **Image** et **ImageNG** modifiée, il suffit de compiler et relancer tous les jeux de tests précédents (Test1.cpp, Test2.cpp, Test3.cpp et Test4.cpp).

b) La classe ImageRGB : les images couleurs

Avant toute chose, on vous demande créer la classe **Couleur** qui représente une couleur et qui comporte

- les variables membres **rouge**, **vert**, **bleu** (**int**) représentant les composantes RGB de la couleur,
- les trois constructeurs classiques et les accesseurs setXXX /getXXX associés aux trois variables membres,
- l'opérateur << permettant d'afficher les caractéristiques de la couleur,
- les 5 variables membres statiques constantes **ROUGE**, **VERT**, **BLEU**, **BLANC** et **NOIR** de type **Couleur** fournissant des objets permanents représentant les couleurs principales citées.

Pour tester cette classe : Test5.cpp, Essai1().

On vous demande à présent de programmer la classe **ImageRGB**, qui hérite de la classe **Image**, et qui présente en plus :

- la variable membre **matrice** de **Couleur** représentant la matrice de pixels de couleur,
- différents constructeurs tels que ceux qui ont été mis en place pour ImageNG,
- les méthodes **Affiche()** et **Dessine()**,
- les méthodes **void setBackground(const Couleur& valeur)**, **void setPixel(int x,int y,const Couleur& valeur)** et **Couleur getPixel(int x,int y)** similaires à celles de ImageNG mais adaptées aux couleurs.
- les méthodes d'instance **void importFromFile(const char* fichier)** et **void exportToFile(const char* fichier, const char* format)**,
- les opérateurs =, << classiques

Pour tester **ImageRGB** : Test5.cpp, Essai2() et Essai3().

c) La classe ImageB : les images binaires

On vous demande à présent de programmer la classe **ImageB**, qui hérite de la classe **Image**, et qui présente en plus :

- la variable membre **matrice** de **bool** représentant la matrice de pixels binaires,
- différents constructeurs tels que ceux qui ont été mis en place pour ImageNG et ImageRGB,
- deux variables membres **couleurTrue** et **couleurFalse** **statiques** et **public** du type **Couleur**. Ces valeurs de couleur seront utilisées pour le dessin (dans la fenêtre graphique) et l'exportation (vers un fichier) de l'image. **couleurTrue** correspond à une valeur **true** du pixel tandis que **couleurFalse** correspond à une valeur **false** du pixel.
- les méthodes **Affiche()** et **Dessine()**,
- les méthodes **void setBackground(bool valeur)**, **void setPixel(int x,int y,bool valeur)** et **bool getPixel(int x,int y)** similaires à celles de ImageNG et Image RGB mais adaptées aux valeurs binaires.
- la méthode **void exportToFile(const char* fichier,const char* format)**. Remarquez qu'une importation à partir d'un fichier n'est pas possible dans le cas des images binaires.
- les opérateurs = et <<

Pour tester **ImageB** et **PixelB** : Test5.cpp, Essai4() et Essai5().

d) Mise en évidence de la virtualité et du down-casting : Essai6() et Essai7()

Normalement rien à programmer ici. …Donc ? Comprendre et être capable d'expliquer le code fourni dans le jeu de test et mettant en évidence le down-casting et le dynamic-cast du C++ → ce sera réutilisé dans l'application finale.

Etape 6 (Test6.cpp) :

Les exceptions

On demande de mettre en place une structure minimale de gestion des erreurs propres aux classes développées jusqu'ici. On va donc imaginer la petite hiérarchie de classes d'exception suivante :

- **Exception** : Cette classe contiendra une seule variable membre du type **chaine de caractères** (**string**). Celle-ci contiendra un message « utilisateur » lié à l'erreur. Cette classe doit comporter les constructeurs et accesseurs classiques et servira de base aux deux classes dérivées décrites ci-dessous.

- **RGBException** : lancée lorsque l'on tente de réaliser une **opération non autorisée sur une couleur ou un niveau de gris**. Cette classe hérite donc de **Exception** et possède en plus une variable membre **valeur** de type **int** contenant la valeur du niveau/composante qui a posé problème. La chaîne de caractères héritée de **Exception** contiendra plus d'informations sur

le sujet comme par exemple « Composante rouge invalide ! » ou encore « Niveau de gris invalide ! ». Cette exception sera lancée lorsqu'on tente de
  - créer/modifier une **Couleur** avec une/des composante(s) non comprise(s) entre 0 et 255.
  - modifier un pixel d'une **ImageNG** (méthode setPixel) avec une valeur de niveau de gris non valide.

- **XYException** : lancée lorsque l'on tente de réaliser une **opération non autorisée sur une dimension ou sur une position de pixel**. Cette classe hérite donc de **Exception** et possède en plus une variable **coordonnee** de type **char** pouvant prendre la valeur **'x'** si l'erreur ne correspond qu'à l'axe x, **'y'** si l'erreur ne correspond qu'à l'axe y ou **'d'** si l'erreur correspond aux deux axes simultanément. La chaîne de caractères héritée de Exception contiendra plus d'informations sur le sujet comme par exemple « Dimension invalide ! » ou encore « Coordonnees pixel invalides ! ». Cette exception sera lancée lorsqu'on tente de
  - créer/modifier un objet de la classe **Dimension** avec une largeur et/ou une hauteur invalide (c'est-à-dire plus petite que 1)
  - comparer des images en niveau de gris **ImageNG** qui n'ont pas les mêmes dimensions (opérateurs <, > et ==).
  - modifier une image en un pixel (méthodes setPixel et opérateurs +) dont les coordonnées sont invalides (c'est-à-dire qui sort de l'image).

Le fait d'insérer la gestion d'exceptions implique qu'elles soient récupérées et traitées lors des tests effectués en première partie d'année (**il faudra donc compléter le jeu de tests Test6.cpp** → utilisation de **try**, **catch** et **throw**), mais également dans l'application finale.

Etape 7 (Test7.cpp) : Les containers et les templates

a) L'utilisation future des containers

On conçoit sans peine que notre future application va utiliser des containers mémoire divers qui permettront par exemple de contenir les images, les couleurs, … Nous allons donc ici mettre en place nos containers pour la suite de l'application.

b) Un container hybride : la classe ArrayList

Il s'agit en fait d'une liste chaînée dynamique mais qui pourra être manipulée comme un vecteur via des indices. Pour rappel, une liste chaînée dynamique présente un pointeur de tête et une succession de cellules liées entre elles par des pointeurs, la dernière cellule pointant vers NULL. Cette liste va être encapsulée dans une **classe ArrayList template** contenant comme seule variable membre le pointeur de tête de la liste chaînée. Elle aura donc la structure de base suivante :

```
template<class T> class Liste
{
	private :
		Cellule<T> *pTete ;
	…
}
```

où les cellules de la liste chaînée auront la structure suivante :

```
template<class T> struct Cellule
{
	T valeur ;
	Cellule<T> *suivant ;
}
```

La classe **ArrayList** devra disposer des méthodes suivantes :

- Un **constructeur par défaut** permettant d'initialiser le pointeur de tête à NULL.
- Un **constructeur de copie**.
- Un **destructeur** permettant de libérer correctement la mémoire.
- La méthode **estVide()** retournant le booléen true si la liste est vide et false sinon.
- La méthode **getNombreElements()** retournant le nombre d'éléments présents dans la liste.
- La méthode **Affiche()** permettant de parcourir la liste et d'en afficher chaque élément.
- La méthode **void insereElement(const T & val)** permettant d'insérer un nouvel élément **à la fin de la liste**.
- La méthode **&T getElement(int ind)** qui permet de récupérer l'élément situé à l'indice **ind** dans la liste mais **sans le supprimer**. Les valeurs possibles pour ind sont 0, …, getNombreElements()-1.
- La méthode **T retireElement(int ind)** qui permet de supprimer et de retourner l'élément situé à l'indice **ind** dans la liste. Les valeurs possibles pour ind sont 0, …, getNombreElements()-1.
- Un **opérateur =** permettant de réaliser l'opération « liste1 = liste2 ; » sans altérer la liste2 et de telle sorte que si la liste1 est modifiée, la liste2 ne l'est pas et réciproquement.

Dans un premier temps, vous testerez votre classe **ArrayList** avec des **entiers**, puis ensuite avec des objets de la classe **Couleur**.

Bien sûr, on travaillera, comme d'habitude, en fichiers séparés afin de maîtriser le problème de l'instanciation des templates.

c) Parcourir et récupérer les éléments d'une liste : l'itérateur de liste

Dans l'état actuel des choses, nous pouvons ajouter/supprimer des éléments à une liste mais nous n'avons aucun moyen simple et standardisé de le parcourir, élément par élément. La notion d'itérateur va nous permettre de réaliser cette opération.

On vous demande donc de créer la classe **Iterateur** qui sera un **itérateur** de la classe **ArrayList** et qui comporte, au minimum, les méthodes et opérateurs suivants :

- **reset()** qui réinitialise l'itérateur au début de la liste.
- **end()** qui retourne le booléen true si l'itérateur est situé au bout de la liste.
- **Opérateur ++** qui déplace l'itérateur vers la droite.
- **Opérateur de casting ()** qui retourne (par valeur) l'élément pointé par l'itérateur.

- **Opérateur T& operator&()** qui retourne (par référence) l'élément pointé par l'itérateur.

On vous demande donc d'utiliser la classe **Iterateur** afin de vous faciliter l'accès au container. Son usage correct sera vérifié lors de l'évaluation finale.

Etape 8 (Test8.cpp)

Première utilisation des flux

Il s'agit ici d'une première utilisation des flux en distinguant les flux caractères et **les flux bytes (méthodes write et read**). Dans cette première approche, nous ne considérerons que les flux bytes.

Les classes ImageNG, ImageB et ImageRGB se sérialisent elles-mêmes

On souhaite enregistrer (« sérialiser ») sur disque les différentes images de notre future application mais dans un format propriétaire (c'est-à-dire propre et uniquement manipulable par notre application). Cela servira pour la sauvegarde de l'état de notre future application.

On demande donc de compléter les classes **ImageNG**, **ImageB** et **ImageRGB** avec les deux méthodes suivantes :

- **Save(ofstream & fichier) const** permettant d'enregistrer sur flux fichier toutes les données d'une image (id, nom, dimension et matrice de pixels) et cela champ par champ. Le fichier obtenu sera un fichier **binaire** (utilisation des méthodes **write** et **read**).

- **Load(ifstream & fichier)** permettant de charger toutes les données relatives à une image enregistrée sur le flux fichier passé en paramètre.

Afin de vous aider dans le développement, on vous demande d'utiliser l'encapsulation, c'est-à-dire de laisser chaque classe gérer sa propre sérialisation. En d'autres termes, on vous demande d'ajouter aux classes **Dimension**, **Image**, et **Couleur** les méthodes suivantes :

- **void Save(ofstream & fichier) const** : méthode permettant à un objet de s'écrire lui-même sur le flux fichier qu'il a reçu en paramètre.
- **void Load(ifstream & fichier)** : méthode permettant à un objet de se lire lui-même sur le flux fichier qu'il a reçu en paramètre.

Ces méthodes seront appelées par les méthodes Save et Load des classes ImageNG/ImageB/ImageRGB lorsqu'elle devra enregistrer ou lire ses variables membres dont le type n'est pas un type de base.

Important

Tous les enregistrements seront de taille variable. Pour l'enregistrement d'une **chaine de caractères** « chaine » de type **char ***, on enregistrera tout d'abord le **nombre de caractères** de la chaine (strlen(chaine)) puis ensuite la **chaine elle-même**. Ainsi, lors de la lecture dans le fichier, on

lit tout d'abord la taille de la chaine et on sait directement combien de caractères il faut allouer et lire ensuite.

Pour les **chaînes de caractères** de type **string**, il faut également faire attention car un objet de type string a une taille pouvant varier et un code du genre

```
string chaine = « Je suis la chaine a enregistrer »;
fichierOut.write((char*)&chaine,sizeof(string));
…
string chaineLue;
fichierIn.read((char*)&chaineLue,sizeof(string));
```

ne fonctionnera pas. Il est nécessaire, comme pour les **char*** d'enregistrer d'abord le nombre de caractères de la chaîne puis la chaîne elle-même :

```
int taille = chaine.size();
fichierOut.write((char*)&taille,sizeof(int));
fichierOut.write((char*)chaine.data(),taille*sizeof(char));
…
int t;
fichierIn.read((char*)&t,sizeof(int));
chaine2.resize(t);   // preuve que le type chaine est de taille variable…
fichierIn.read((char*)chaine2.data(),t*sizeof(char));
```

Etape 9 (Test9.cpp) :

Traitements d'images simples : Création de méthodes statiques

Nous allons à présent mettre en place quelques techniques élémentaires de traitements d'images. Chacune de ces techniques correspondra à une fonction prenant en entrée une image (et éventuellement des paramètres) et qui fournira en sortie l'image traitée. On vous demande de créer la classe **Traitements** ne contenant que les **méthodes statiques** décrites ci-dessous.

a) Seuillage d'une ImageNG (Essai1())

Le seuillage d'une image en niveaux de gris consiste à classifier les pixels en séparant les pixels « clairs » des pixels « foncés » (on pourrait donc par exemple séparer les objets foncés du fond clair d'une image). On vous demande donc de programmer la méthode statique

ImageB Seuillage(const ImageNG& imageIn, int seuil) ;

L'image de sortie est une image binaire, de même taille que l'image d'entrée (imageIn), et dont les valeurs de pixels valent

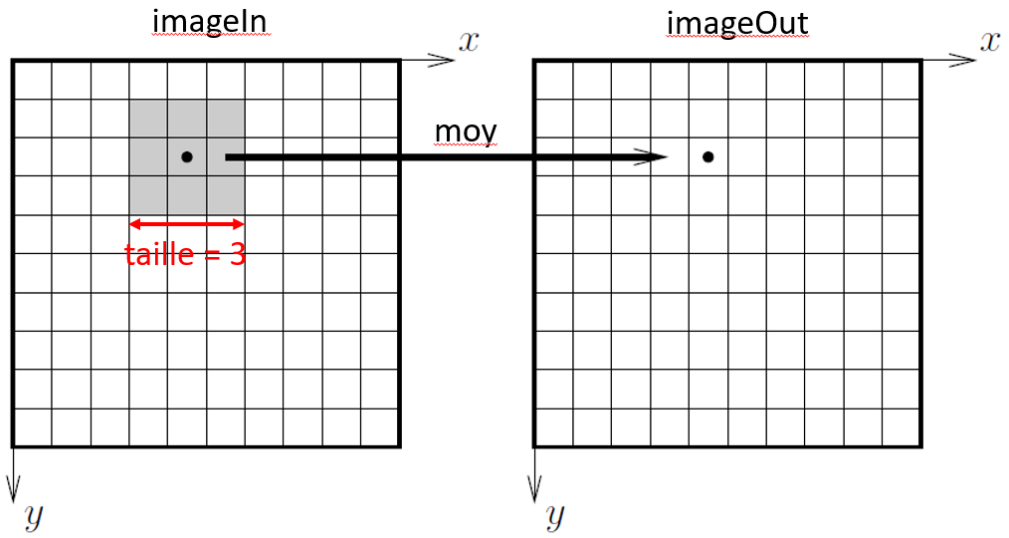
- **true** si le niveau de gris du pixel correspondant de imageIn est supérieur au **seuil**,

- **false** si le niveau de gris du pixel correspondant de imageIn est inférieur au **seuil**.

Si le nom de l'image d'entrée est « lena » et que l'on réalise un seuillage à 100, le nom de l'image de sortie pourrait être « lena-seuil100 ».

b) Filtre moyenneur d'une ImageNG (Essai2())

Le filtre moyenneur appliqué à une image en niveaux de gris consiste à remplacer chaque pixel de l'image traitée par la **moyenne** des valeurs de pixels entourant ce pixel. Ceci est illustré à la figure suivante :



Dans cet exemple, la taille du filtre est égale à 3, cela consiste à faire la moyenne des 3×3 = 9 pixels situés autour du pixel traité •. La taille du filtre sera toujours un nombre impair, ce qui correspond à moyenner les 3×3 = 9, 5×5 = 25, 7×7 = 49, … pixels situés autour du pixel traité.

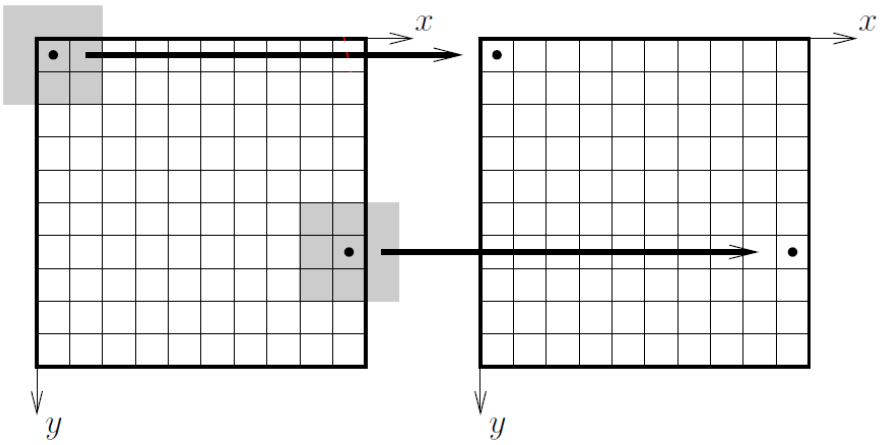
L'effet de ce filtre est de flouter l'image. On vous demande donc de programmer la méthode statique

ImageNG FiltreMoyenneur(const ImageNG& imageIn, int taille) ;

L'image de sortie est une ImageNG de même taille que imageIn. Le paramètre taille correspond à la taille du filtre (3, 5, 7, …). Si le nom de l'image d'entrée est « lena » et que l'on applique ce filtre avec une taille 5, le nom de l'image de sortie pourrait être « lena-moyenne5 ».

Remarque importante

Que se passe-t-il lorsque les pixels utilisés pour calculer une moyenne se situent en dehors de l'image ?



Vous avez le choix entre deux possibilités :

- les pixels extérieurs sont supposés égaux à la valeur 0.
- les moyennes se font sur moins de pixels. Sur l'exemple de la figure ci-dessus (filtre de taille 3), la moyenne pour le pixel en haut à gauche se fait sur 4 pixels, tandis que la moyenne sur l'autre pixel se fait sur 6 pixels.

c) Filtre médian d'une ImageNG (Essai3())

Le filtre médian d'une image en niveaux de gris est très similaire au filtre moyenneur. Le processus de traitement est identique à la différence près que le pixel traité est remplacé par la **médiane** des valeurs des pixels situés autour du pixel traité. L'utilisation de ce filtre est particulièrement appréciée lorsque l'on veut supprimer des parasites du type « bruit poivre et sel » présents dans l'image (voir l'image « house.bmp » fournie).

Pour rappel, la médiane d'une série de nombres est la valeur située au « milieu » de toutes ces valeurs. Par exemple, la médiane des valeurs {95, 112, 156, 175, 175} est 156. Pour traiter un pixel, il vous suffira de trier une liste de nombres dans laquelle vous insérerez les valeurs des 3×3 = 9, 5×5 = 25, 7×7 = 49, … pixels situés autour du pixel traité, et d'en extraire la valeur centrale.

Bonus : Afin de trier cette liste de nombres, on pourrait imaginer programmer une classe **SortedArrayList**, héritant de **ArrayList**, en permanence triée et qui surchargerait la méthode **insereElement()** de **ArrayList** afin que l'insertion se fasse au bon endroit.

On vous demande donc de programmer la méthode statique

ImageNG FiltreMedian(const ImageNG& imageIn, int taille) ;

L'image de sortie est une ImageNG de même taille que imageIn. Le paramètre taille correspond à la taille du filtre. Si le nom de l'image d'entrée est « lena » et que l'on applique ce filtre avec une taille 5, le nom de l'image de sortie pourrait être « lena-median5 ».

d) Erosion et dilatation d'une ImageNG (Essai4, 5 et 6())

Les opérations d'érosion et dilatation sur une ImageNG sont à nouveau similaire au filtre médian et au filtre moyenneur. Le processus de traitement est identique à la différence que le pixel traité est remplacé par le **minimum** (dans le cas d'une **érosion**) / **maximum** (dans le cas d'une **dilatation**) des valeurs des pixels situés autour du pixel traité. Utilisées seules, ces opérations n'ont pas grande utilité. C'est la combinaison de plusieurs d'entre elles qui devient utile, comme par exemple la détection des contours des objets présents dans l'image (Essai6()). On vous demande donc de programmer les méthodes statiques

ImageNG Erosion(const ImageNG& imageIn, int taille) ;
ImageNG Dilatation(const ImageNG& imageIn, int taille) ;

L'image de sortie est une ImageNG de même taille que imageIn, et le paramètre taille correspond toujours à la taille du filtre. Si le nom de l'image d'entrée est « lena » et que l'on applique ce filtre avec une taille 5, le nom de l'image de sortie pourrait être « lena-erode5 » ou « lena-dilate5 ».

e) Négatif d'une ImageNG (Essai7())

Le négatif d'une image en niveaux de gris consiste à remplacer chaque valeur **X** de pixel par son « complément », c'est-à-dire par la valeur **(255-X)**. Tous les pixels de mêmes niveaux de gris sont donc remplacés par la même valeur indépendamment de leurs voisins. Cette technique entre dans une famille plus générale de techniques appelées « manipulations d'histogramme » qui servent à augmenter le contraste d'une image. On vous demande donc de programmer la méthode statique

ImageNG Negatif(const ImageNG& imageIn);

Si le nom de l'image d'entrée est « lena » et que l'on applique ce traitement, le nom de l'image de sortie pourrait être « lena-negatif ».

Il existe bien d'autres techniques de traitement d'images mais nous en resterons là pour cette incursion dans le domaine.

Etape 10 (Test10a.cpp et Test10b.cpp) :

Un conteneur pour toutes nos données : La classe PhotoShop

a) Mise en place de la classe PhotoShop (Test10a.cpp)

Afin d'éviter que les données de l'application soient dispersées et déclarées à différents endroits du code, on vous demande de regrouper toutes ces données au sein de la même classe appelée **PhotoShop** :

```
class PhotoShop
{
  private:
    ArrayList<Image*> images;
    static int numCourant;

  public:
    PhotoShop();
    ~PhotoShop();

    void   reset();
    void   ajouteImage(Image* pImage);

    void   afficheImages() const;
    void   dessineImages() const;

    Image* getImageParIndice(int indice);
    Image* getImageParId(int id);

    void   supprimeImageParIndice(int ind);
    void   supprimeImageParId(int id);
};
```

Cette classe comporte dans un premier temps :

- La liste des images (**ArrayList<Image*>** **images**) conservées en mémoire dans l'application en vue de traitements ultérieurs ou d'opérations entre images. C'est en quelque sorte la bibliothèque d'images disponibles → l'ArrayList ne contiendra pas d'objet « Image » mais des pointeurs « génériques » (de type **Image***) pouvant pointer vers des ImageNB, des ImageRGB ou des ImageB → un seul conteneur suffit donc à référencer toutes les images en mémoire quel que soit leur type.

- Un **constructeur par défaut**. Inutile de faire un constructeur de copie car il n'existera qu'un seul objet instance de la classe **PhotoShop** dans l'application.

- La méthode **void ajouteImage(Image* pImage)** qui permet d'ajouter au conteneur **images** une image qui a été allouée dynamiquement au préalable. C'est également cette méthode qui est responsable d'attribuer un identifiant (id) unique à chaque image. Avant d'insérer pImage dans le conteneur **images**, la méthode lui assigne comme id la valeur de la **variable statique numCourant** (qui est au préalable initialisée à 1) avant de l'incrémenter de 1.

- Les méthodes **void afficheImages()** et **void dessineImages()** qui permettent d'afficher en console / dessiner les images contenues dans le conteneur **images** → utilisation de l'itérateur et des méthodes virtuelles **Affiche()** et **Dessine()** de la classe Image.

- Les méthodes **getImageParIndice(int indice)** et **getImageParId(int id)** qui permettent de retourner le pointeur vers l'image située dans le conteneur **images** correspondant à l'indice / id passé en paramètre.

- Les méthodes **supprimeImageParIndice(int indice)** et **supprimeImageParId(int id)** qui permettent de supprimer du conteneur **images** l'image dont l'indice / id est passé en paramètre à la fonction → attention que la cellule de l'ArrayList doit être supprimée (ce que fait la méthode retireElement()) mais il faut également désallouer l'image retirée du conteneur (delete) !

- La méthode **reset()** permet de supprimer (et désallouer) toutes les images du conteneur **images** et de remettre la variable statique **numCourant** à 1 → cette méthode devrait être appelée par le destructeur de la classe **PhotoShop**.

b) La classe PhotoShop en tant que singleton (Test10b.cpp)

La classe **PhotoShop** contenant toutes les données de notre future application, on se rend bien compte qu'elle ne sera **instanciée qu'une seule fois**. Une telle classe est appelée un « **singleton** ».

« Le **singleton** est un **patron de conception** (« **Design pattern** ») dont l'objectif est de restreindre l'instanciation d'une classe à un seul objet. On fournira un accès global à cet objet. Il est utilisé lorsqu'on a besoin d'exactement un objet pour coordonner les opérations d'un système. » (wikipedia)

En C++ (et donc dans notre cas pour la classe **PhotoShop**), une manière de faire est

- Rendre le **constructeur par défaut privé** : cela empêchera un programmateur d'instancier un ou plusieurs objets de classe PhotoShop.
- Placer dans la classe PhotoShop une **variable statique privée de type PhotoShop**, que l'on appellera **instance** (il s'agira de l'unique instance de la classe PhotoShop). Cet objet sera instancié en utilisant le constructeur par défaut.
- Cet objet étant privé, on lui donnera un accès « global » en écrivant une **méthode statique publique PhotoShop& getInstance()** qui retourne une référence vers l'objet **instance** qui pourra donc être manipulé de n'importe où dans l'application.
- Afin d'éviter qu'un programmateur puisse obtenir une copie de cette instance, on déclarera en **privé** (mais on ne les définira pas dans PhotoShop.cpp !!) un **constructeur de copie** et un **opérateur =** dans PhotoShop.h.

Remarquez que lorsque vous aurez terminé les modifications de la classe **PhotoShop** afin de la transformer en singleton, le programme Test10a.cpp ne compilera plus. (Pourquoi 😊 ?)

Etape 11 (utilisation de InterfaceQt.tar fourni)

Mise en place de l'interface graphique : Introduction à Qt

a) Introduction

Nous allons à présent mettre en place l'interface graphique. Celle-ci sera construite en utilisant la librairie graphique Qt. Sans entrer dans les détails de cette librairie, il faut savoir qu'une **fenêtre** est représentée par une **classe** dont

- les variables membres sont les composants graphiques apparaissant dans la fenêtre : les boutons, les champs de texte, les checkbox, les tables, …
- les méthodes publiques permettent d'accéder à ses composants graphiques, soit en récupérant les données qui sont encodées par l'utilisateur, soit en y insérant des données.

L'interface graphique de notre application sera la classe **MainWindowPhotoShop** qui a été construite, pour vous, à l'aide de l'IDE QtCreator. Cette classe est fonctionnelle mais il ne s'agit que d'une **coquille vide** qui va permettre de manipuler **le seul objet de classe PhotoShop** de notre application. La classe **MainWindowPhotoShop** est fournie par les fichiers

- **mainwindowphotoshop.h** qui contient la définition de la classe et la déclaration de ses méthodes
- **mainwindowphotoshop.cpp** qui contient la définition de ses méthodes.

Seuls ces deux fichiers pourront être modifiés par vous. Les autres fichiers fournis (dans InterfaceQt.tar) ne pourront en aucun cas être modifiés. Le main de votre application est le fichier main.cpp également fourni et ne devra pas être modifié. Un makefile est également fourni pour la compilation. A vous de combiner votre makefile actuel et le makefile fourni. Dès que vous aurez combiné ces deux makefiles, n'oubliez pas de décommenter (dans les fichiers ci-dessus) les **méthodes** et les **#include** faisant intervenir les classes que vous avez développées.

La classe **MainWindowPhotoShop** contient déjà un ensemble de méthodes fournies (et que vous ne devez donc pas modifier) afin de vous faciliter l'accès aux différents composants graphiques.

Les méthodes que vous devez modifier contiennent le commentaire // TO DO. Elles correspondent aux différents boutons et items de menu de l'application et correspondent toutes à une action demandée par l'utilisateur, comme par exemple « Charger ImageNG », « Enregistrer ImageRGB », …

Pour l'affichage de messages dans des **boîtes de dialogue** ou des saisies de int/float/chaînes de caractères/couleur via des boîtes de dialogues, vous disposez des méthodes (**que vous ne devez pas modifier mais juste utiliser**) :

- void dialogueMessage(const char* titre,const char* message);
- void dialogueErreur(const char* titre,const char* message);
- string dialogueDemandeTexte(const char* titre,const char* question);
- int dialogueDemandeInt(const char* titre,const char* question);
- float dialogueDemandeFloat(const char* titre,const char* question);

- string dialogueDemandeFichierOuvrir(const char* question);
- string dialogueDemandeFichierEnregistrer(const char* question);
- void dialogueDemandeCouleur(const char* message,int* pRouge,int* pVert,int* pBleu);

La dernière méthode reçoit l'adresse de 3 variables qui recevront les composantes RGB d'une couleur saisie par l'utilisateur.

Etant une classe C++ au sens propre du terme, vous pouvez ajouter, à la classe **MainWindowPhotoShop**, des variables et des méthodes membres en fonction de vos besoins.

b) Mise en place des premières fonctionnalités

Les premières fonctionnalités demandées ici correspondent

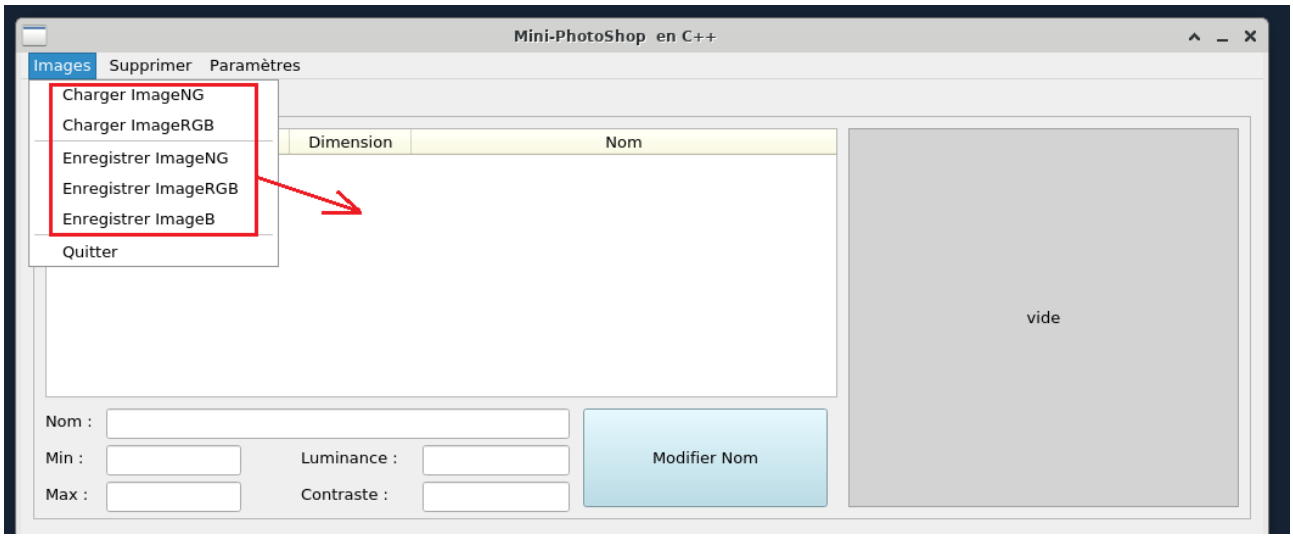
- au chargement d'images NG et RGB à partir d'images au format JPG, BMP ou PNG afin de les importer dans la bibliothèque de l'application
- à l'enregistrement sur disque au format JPB, BMP ou PNG d'images de la bibliothèque de l'application
- à la suppression d'images de la bibliothèque soit par sélection dans la table, soit par id
- à la visualisation d'une image sélectionnée dans la zone supérieure droite de la fenêtre, à l'affichage de ses paramètres (min, max, luminance et contraste) dans le cas d'une ImageNG, et à la modification de son nom.

Les 3 premières fonctionnalités ont déjà été programmées à l'étape précédente dans la classe **PhotoShop**. Il suffit donc de manipuler l'unique instance statique de cet objet à l'aide de l'interface graphique. Toute la logique de la gestion de la bibliothèque d'images de l'application, dite « logique métier », se trouve dans la classe **PhotoShop**.

Etant un objet statique, l'unique instance de la classe **PhotoShop** sera accessible partout dans l'application à partir de la classe **PhotoShop** elle-même :

- **PhotoShop& PhotoShop::getInstance()** : qui retourne la référence vers le singleton de notre application.

Les fonctionnalités correspondant aux 2 premiers items demandés ici correspondent à l'encadré rouge ci-dessous :



Il s'agit donc de modifier les méthodes suivantes de la classe **MainWindowPhotoShop** :

- void on_actionCharger_ImageNB_triggered();
- void on_actionCharger_ImageRGB_triggered();
- void on_actionEnregistrer_ImageNB_triggered();
- void on_actionEnregistrer_ImageRGB_triggered();
- void on_actionEnregistrer_ImageB_triggered();

pour les items de menu « Images ».

Pour **charger une image** en cliquant sur un des items de menu « Charger imageXXX » correspondant, vous devez (dans la méthode **on_actionCharger_ImageXXX_triggered**)

1. demander à l'utilisateur le nom du fichier de l'image à charger. Pour cela, vous disposez de la méthode **string dialogueDemandeFichierOuvrir(const char* question)**.
2. instancier de manière dynamique (**new**) un objet de la classe **Image** correspondant, charger le fichier en utilisant la méthode **importFromFile()** de l'image et l'insérer dans la bibliothèque en appelant la méthode **void ajouteImage(Image* pImage)** du singleton de la classe **PhotoShop**.
3. Mettre à jour la table des images. Pour cela, vous devez
   a. Vider la table des images à l'aide de la méthode **void videTableImages()**.
   b. Récupérer la liste des images. L'**itérateur** vous permet de parcourir cette liste et d'en récupérer chaque élément. Pour chaque image, vous devez utiliser la méthode **void ajouteTupleTableImages(int id,string type,string dimension,string nom)** qui ajoute une nouvelle ligne à la table. Le type correspond à « NG », « RGB » ou « B ».

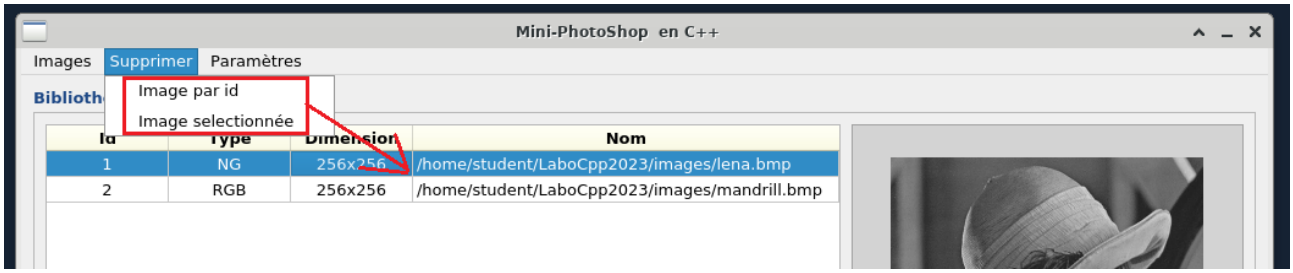
Pour **enregistrer une image** en cliquant sur un des items de menu « Enregistrer imageXXX » correspondant, vous devez (dans la méthode **on_actionEnregistrer_ImageXXX_triggered**)

1. récupérer l'indice de l'image sélectionnée dans la table. Pour cela vous disposez de la méthode **int getIndiceImageSelectionnee()** qui retourne l'indice dans la table de l'image

sélectionnée (il s'agit également de son indice dans la liste images de la classe **PhotoShop**), et -1 si aucune image n'est sélectionnée. Si aucune image n'est sélectionnée, vous pouvez afficher une boîte de dialogue d'erreur en utilisant la méthode **void dialogueErreur(const char* titre,const char* message)**.

2. Vérifier que le type d'image (NG, RGB ou B) correspond à l'item de menu choisi et dans le cas contraire afficher un message d'erreur.
3. demander à l'utilisateur le nom du fichier de l'image à enregistrer. Pour cela, vous disposez de la méthode **string dialogueDemandeFichierEnregistrer(const char* question)**.
4. Demander à l'utilisateur le format du fichier (« JPG », « BMP » ou « PNG). Pour cela, vous disposez de la méthode **string dialogueDemandeTexte(const char *titre, const char *question)**
5. Enregistrer l'image sur disque en utilisant la méthode **exportToFile()** de l'image.

La **suppression d'une image** correspond à l'encadré rouge ci-dessous :



Il s'agit donc de modifier les méthodes suivantes de la classe **MainWindowPhotoShop** :

- void on_actionImage_selectionn_e_triggered();
- void on_actionImage_par_id_triggered();

pour les items de menu « Supprimer ».

Le clic sur **l'item de menu « Image par id » du menu « Supprimer » supprime une image dont on demandera l'id à l'utilisateur**. Pour ce faire, vous devez (dans la méthode void on_actionImage_par_id_triggered) :

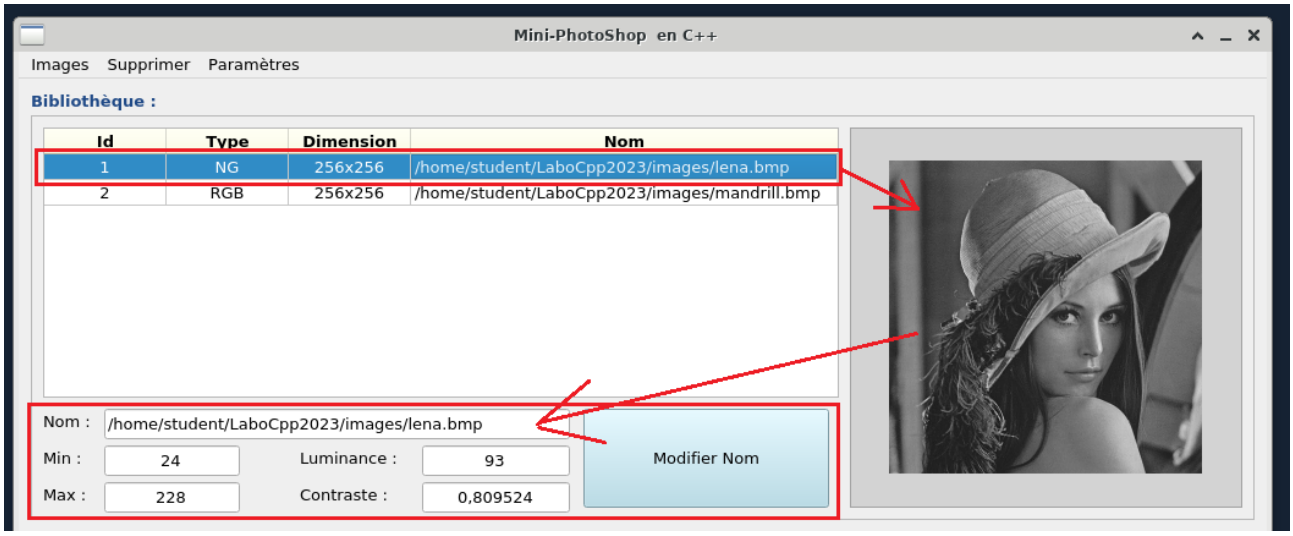
1. Demander à l'utilisateur d'encoder un id. Pour cela, vous devez afficher une boîte de dialogue de saisie d'entier en utilisant la méthode **int dialogueDemandeInt(const char* titre,const char* question)**.
2. Appeler la méthode **void supprimeImageParId(int id)** de l'objet **PhotoShop**.
3. Mettre à jour la table des images (voir ci-dessus).

Le clic sur le **l'item de menu « Image sélectionnée » du menu « Supprimer » supprime l'image sélectionnée dans la table**. Pour ce faire, vous devez (dans la méthode void on_actionImage_selection_e_triggered) :

1. Récupérer l'indice de l'image sélectionnée dans la table. Pour cela vous disposez de la méthode **int getIndiceImageSelectionnee()** qui retourne l'indice dans la table de l'image sélectionnée (il s'agit également de son indice dans le conteneur **images** de **PhotoShop**), et -1 si aucune image n'est sélectionnée. Si aucune image n'est sélectionnée, vous pouvez afficher une boîte de dialogue d'erreur en utilisant la méthode **void dialogueErreur(const char* titre,const char* message)**.

2. Appeler la méthode **void supprimeImageParIndice(int ind)** de l'objet **PhotoShop**.
3. Mettre à jour la table des images (voir ci-dessus).

La **visualisation d'une image** (et de ses paramètres dans le cas d'une ImageNG) correspond à l'encadré rouge ci-dessous :



Il s'agit donc de modifier la méthode suivante de la classe **MainWindowPhotoShop** :

- void on_tableWidgetImages_itemSelectionChanged();

qui est appelée lorsque l'on clique (et donc sélectionne) une image dans la table.

Dans cette méthode, vous devez

1. récupérer l'indice de l'image sélectionnée (voir plus haut)
2. récupérer l'image correspondante dans l'objet **PhotoShop** et placer son nom dans l'interface graphique à l'aide de la méthode **void setNomImage(string nom)**
3. si le type de l'image est NG, vous devez utiliser la méthode **void setParametresImageNG(int max=-1,int min=-1,int luminance=-1,float contraste=0.0f)** pour afficher ses paramètres dans l'interface graphique. Les paramètres par défaut de cette méthode permettent de vider les champs lorsque l'image n'est pas de type NG.
4. Pour dessiner l'image dans la zone droite de la fenêtre, vous devez utiliser une des méthodes
   i. void setImageNG(string **destination**,const ImageNG* imageng=NULL);
   ii. void setImageRGB(string **destination**,const ImageRGB* imagergb=NULL);
   iii. void setImageB(string **destination**,const ImageB* imageb=NULL);

   selon le type d'image. **destination** est une chaine de caractères qui doit valoir **« selection »** ici. Si le paramètre par défaut (NULL) est utilisé, la zone correspondant à l'image est vidée.

Reste à gérer le bouton **« Modifier Nom »** qui permet de modifier le nom de l'image actuellement sélectionnée. L'utilisateur doit simplement modifier le nom dans la zone correspondante de la fenêtre et cliquer sur le bouton. Vous devez donc de modifier la méthode suivante de la classe **MainWindowPhotoShop** :

- void on_pushButtonModifierNom_clicked();

qui est appelée lorsque l'on clique sur ce bouton. Dans cette méthode, vous devez

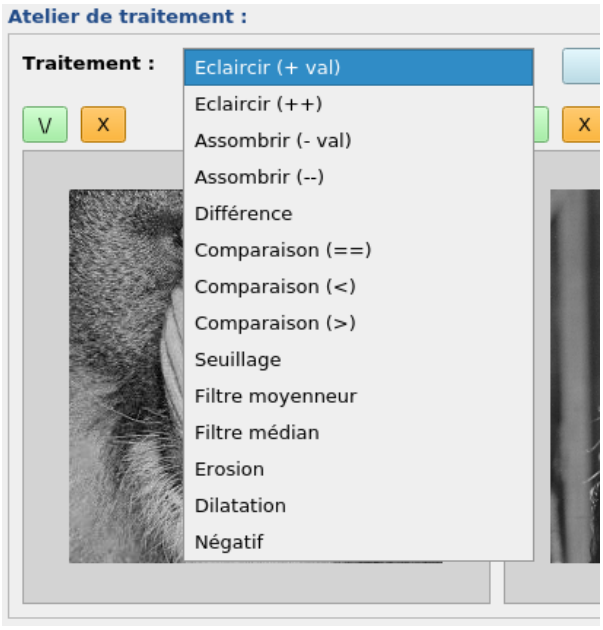
- récupérer le nom modifié par l'utilisateur. Pour cela, vous disposez de la méthode **string getNomImage() const**
- récupérer l'indice de l'image sélectionnée ainsi que l'image correspondante du singleton
- appeler la méthode **setNom()** de l'image et mettre à jour la table des images.

Etape 12 :

Les traitements d'image via l'interface Qt

Nous allons à présent compléter les fonctionnalités de l'interface graphique de notre application, et plus particulièrement les **traitements d'images** réalisés via les composants graphiques situés en bas de la fenêtre graphique :

Les différents traitements correspondent à tous ceux qui ont déjà été développés aux étapes 4 et 9, à savoir



L'opération à réaliser se choisit via la comboBox qui a été prérempli par la méthode **void ajouteTraitementDisponible(string operation)**. Certaines opérations ne nécessitent **qu'une seule opérande** « image » (comme le seuillage, l'opérateur ++, …) et d'autres en nécessitent **deux** (comme la soustraction, l'opérateur ==, …). D'où les 2 opérandes que l'on voir apparaître sur l'interface graphique.

Le **résultat** d'une opération peut être une **image** (celle à droite intitulée « Résultat ») ou un **booléan** lorsqu'il s'agit d'une comparaison. Le résultat booléen d'une opération sera affiché via la checkbox intitulé actuellement « Résultat » et coloré en jaune.

Pour **visualiser une image** dans une des 3 zones prévues, vous disposez des méthodes

- **void setImageNG(string destination,const ImageNG* imageng=NULL);**
- **void setImageRGB(string destination,const ImageRGB* imagergb=NULL);**
- **void setImageB(string destination,const ImageB* imageb=NULL);**

déjà abordées plus haut, et où

- **destination** est une chaine de caractères pouvant valoir
  - **« selection »** si vous voulez dessiner l'image en haut à droite
  - **« operande1 »** si vous voulez dessiner l'image en bas en tant qu'opérande 1
  - **« operande2 »** si vous voulez dessiner l'image en bas en tant qu'opérande 2
  - **« resultat »** si vous voulez dessiner l'image en bas en tant que résultat
- Le pointeur de type ImageXXX est un pointeur vers l'image à dessiner → dans le cas où ce pointeur est NULL, aucune image n'est dessinée et la zone est remplie avec la chaine de caractères « Vide »

Pour **visualiser un résultat booléen**, vous disposez de la méthode **void setResultatBoolean(int valeur=-1)** qui affiche

- TRUE en vert si valeur est égal à 1
- FALSE en rouge si valeur est égal à 0
- Résultat en jaune si valeur est égal à -1

L'interface graphique n'a **aucune mémoire des images qu'elle contient** et c'est à vous de mémoriser les adresses vers les images « opérandes » et « résultat ». Pour cela, on vous demande d'ajoute à la classe **PhotoShop**

- **Deux variables membres statiques publiques de type Image*, operande1 et operande2**, valant NULL ou pointant vers une des images de la bibliothèque (donc vers une des images de la liste **images** de **PhotoShop**)
- **Une variable membre statique publique de type Image*, resultat**, qui vaudra NULL (si pas de résultat actuellement) ou pointant vers une image « résultat d'une opération » et ayant été allouée dynamiquement. **Remarquez que resultat ne pointera jamais vers une des images de la liste images de l'objet PhotoShop**.

Reste maintenant à gérer l'action des différents boutons.

Les **deux boutons verts « \/ »** permettent de sélectionner une des images de la bibliothèque et de la mettre dans la zone correspondant à l'opérande 1 ou 2. Pour cela vous devez compléter les méthodes

- void on_pushButtonOperande1_clicked()
- void on_pushButtonOperande2_clicked()

dans lesquelles vous devez

1. récupérer l'indice de l'image sélectionnée (voir plus haut)
2. récupérer un pointeur vers cette image et l'affecter à la variable operande1 de **PhotoShop**
3. appeler la bonne méthode setImageXXX() en fonction du type de l'image

Remarquez qu'un clic sur un de ces deux boutons ne retire pas l'image correspondante de la bibliothèque.

Les **deux boutons orange « X »** permettent de supprimer l'opérande 1 ou 2 (toujours sans supprimer l'image correspondante de la bibliothèque). Pour cela, vous devez modifier les méthodes :

- void on_pushButtonSupprimeOperande1_clicked()
- void on_pushButtonSupprimeOperande2_clicked()

dans lesquelles vous devez

- remettre la variable operande1 ou operande2 de **PhotoShop** à NULL
- appeler la bonne méthode setImageXXX() afin de faire disparaître l'image « operande »

Attention !!! Ne faites pas de delete de operande1 ou operande2 sous peine de perdre l'image correspondante de la bibliothèque et de provoquer un bug…

Le **bouton « Go ! »** permet de lancer un traitement. Pour cela, vous devez compléter la méthode

- void on_pushButtonTraitement_clicked()

dans laquelle vous devez

1. récupérer le traitement choisi par l'utilisateur. Pour cela, vous disposez de la méthode **string getTraitementSelectionne() const**
2. vérifier si operande1 est non NULL (et si operande2 est non NULL si le traitement nécessite 2 images)
3. vérifier si l'(les) opérande(s) est (sont) du bon type pour le traitement choisi. Ici, nous n'avons considéré que des traitements d'ImageNG (mais on pourrait imaginer d'en considérer d'autres..).
4. Appeler **le bon opérateur** ou la **bonne méthode** de la classe **Traitement**. Si un entier est nécessaire (seuil ou taille du filtre), vous devez utiliser la méthode **int dialogueDemandeInt(const char *titre, const char *question)**
5. L'image résultat **doit être allouée dynamiquement**. Une fois celle-ci obtenue, son adresse est placée dans la variable resultat de **PhotoShop** et elle est dessinée à droite, à l'aide d'une des méthodes **setImageXXX().**

Remarquez que certaines opérations (<, >, ==, -) peuvent provoquer le **lancement d'exceptions** que vous devez gérer proprement.

Un clic sur le **bouton jaune « /\ »** (méthode **void on_pushButtonResultat_clicked()** à modifier) permet de

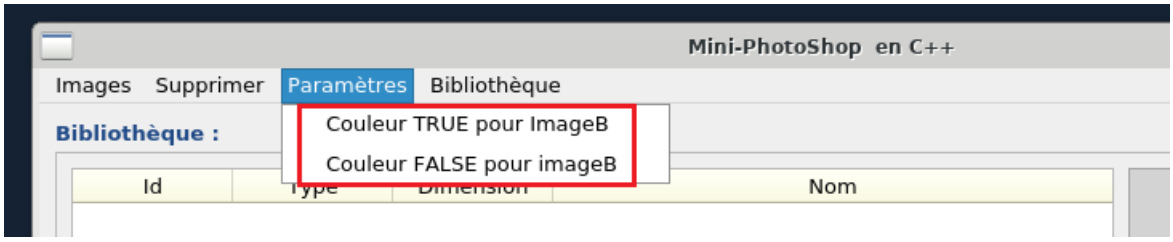
1. Ajouter l'image « résultat » à la bibliothèque (donc dans l'objet **PhotoShop**)
2. Mettre à jour la tables des images
3. Remettre le pointeur resultat de **PhotoShop** à NULL
4. Effacer l'image « résultat » en appelant la méthode **setImageXXX()**.

Un clic sur le **bouton orange « X »** (méthode **void on_pushButtonSupprimerResultat_clicked()** à modifier) de droite permet de

1. Supprimer l'image resultat par un **delete**
2. Remettre le pointeur resultat de **PhotoShop** à NULL
3. Effacer l'image « résultat » en appelant la méthode setImageXXX().

Afin d'avoir un meilleur rendu d'affichage des résultats d'un traitement, il serait judicieux, avant un traitement, de remettre la variable resultat à NULL (avec un delete au préalable), d'effacer l'image résultat s'il y en a une, et de « reseter » le booléen.

Reste une dernière petite chose à mettre en place : le **choix des couleurs à afficher pour TRUE et FALSE** dans le cas des images B :



Pour cela, on vous demande de compléter les méthodes

- void on_actionCouleur_TRUE_pour_ImageB_triggered();
- void on_actionCouleur_FALSE_pour_imageB_triggered();

dans lesquelles vous devez :

1. demander à l'utilisateur de choisir une couleur. Pour cela, vous devez utiliser la méthode **void dialogueDemandeCouleur(const char* message,int* pRouge,int* pVert,int* pBleu)**
2. modifier la variable **couleurTrue** ou **couleurFalse** de la classe **ImageB**

Etape 13 :
Importation d'images à partir d'un fichier csv : un fichier texte

Il serait intéressant de pouvoir importer d'un seul coup un ensemble d'images à partir d'un fichier texte dont le contenu pourrait être

```
NG;../images/lena.bmp;lena
RGB;../images/lena.bmp;lena
…
NG;../images/house.bmp;house
…
RGB;../images/mandrill.bmp;mandrill
```

Chaque ligne de ce fichier correspond à une image qui doit être chargée en mémoire. Il s'agit en fait d'un **fichier csv** dont le caractère **';'** est appelé le ***séparateur***. Celui-ci pourrait être **':'** ou encore **','**. Ce type de fichier peut être ouvert dans n'importe quel éditeur de texte ou Excel afin de pouvoir le modifier. Au laboratoire, on vous fournira le fichier **images.csv**.

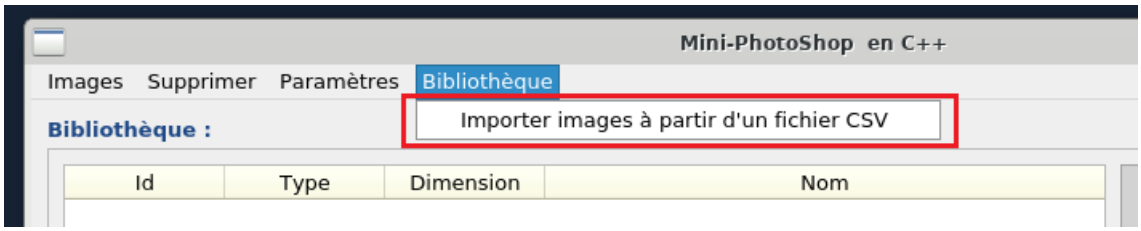
Chaque ligne de ce fichier contient 3 informations :

1. le type d'image que l'on veut importer,
2. le nom (et chemin) du fichier BMP, JPG ou PNG que l'on veut lire.
3. le nom que l'on désire attribuer à l'image dans l'application

On vous demande donc d'ajouter à la classe **PhotoShop** la méthode **int importeImages(string nomFichier)** qui doit

- ouvrir le fichier dont le nom est passé en paramètre.
- lire le fichier ligne par ligne : en fonction du type d'image lu (« NG » ou « RGB »), elle doit créer un objet du bon type, lui donner le nom souhaité et l'insérer dans le conteneur **images** de l'objet **PhotoShop**.
- retourner le nombre d'images correctement importées.

Vous devez ensuite ajouter la fonctionnalité à l'application :



Pour cela, on vous demande compléter la méthode **void on_actionImporterCSV_triggered()** dans laquelle :

1. vous demandez à l'utilisateur d'encoder un nom du fichier CSV. Pour cela, vous pouvez utiliser la méthode **string dialogueDemandeFichierOuvrir(const char* question)**
2. appeler la méthode **importeImages()** de l'objet **PhotoShop**

3. mettre à jour la tables des images
4. afficher dans une boîte de dialogue le nombre d'images qui ont été importées

Etape 14 :
Reset et Sauvegarde de l'état de l'application : un fichier binaire

Lorsque l'on quittera l'application, celle-ci devra demander à l'utilisateur s'il souhaite enregistrer **l'état de la bibliothèque**. Si oui, toutes les données pertinentes de l'application devront être sérialisées dans un **fichier binaire** (utilisation des méthodes **write** et **read**) appelé **« sauvegarde.dat ».**

Un tel fichier sera structuré de la manière suivante :

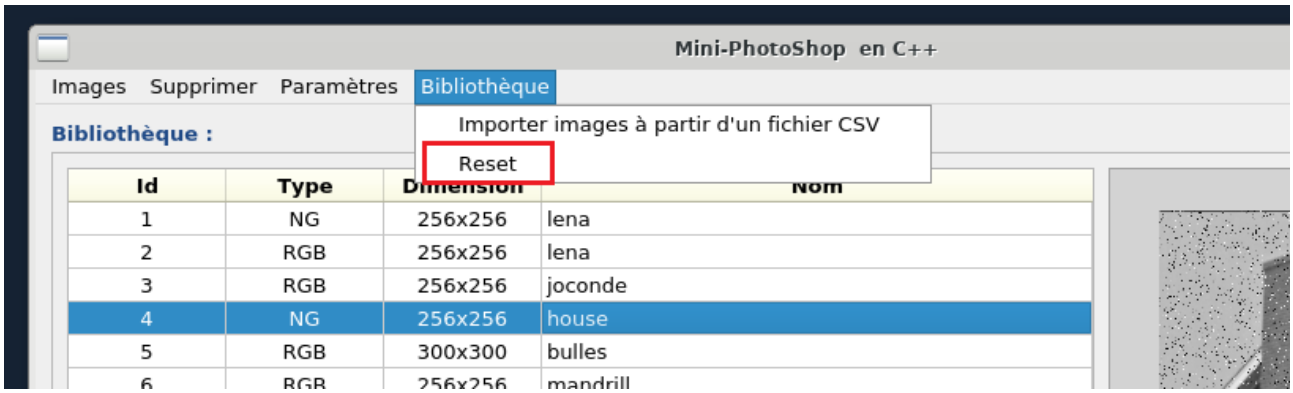
1. la variable (int) **numCourant** de la classe **PhotoShop**
2. les variables **couleurTrue** et **CouleurFalse** de la classe **ImageB**
3. les images de la bibliothèque, donc
    a. un **entier** contenant le nombre d'images dans le conteneur images, puis pour chaque image
    b. un **entier** valant **1 pour ImageNG**, **2 pour ImageRGB**, **3 pour ImageB**
    c. l'image en faisant appel à la méthode **Save()** de la classe ImageXXX

Ne sachant pas à l'avance quel type d'image est écrit sur disque, lors de la relecture, l'entier lu nous informera quel type d'image doit être instancié puis chargé à l'aide de sa méthode **Load().**

On vous demande donc de

- ajouter les méthodes **Save()** et **Load()** à la classe **PhotoShop** afin qu'elle (de)sérialise son état (au format binaire) dans le fichier **« sauvegarde.dat »**, selon la structure expliquée ci-dessus.
- faire en sorte que lorsque l'on quitte l'application (item de menu « Quitter » ou « croix de la fenêtre »), celle-ci enregistre son état dans le fichier binaire **« sauvegarde.dat »**. Pour ce faire vous devez compléter les méthodes **void on_actionQuitter_triggered()** et **void closeEvent(QCloseEvent *event)**
- Au démarrage, l'application doit tenter d'ouvrir le fichier **« sauvegarde.dat »** afin de restaurer l'état de la bibliothèque. Si celui-ci n'existe pas, une bibliothèque « vierge » doit être utilisé.

Pour finir cette étape, on vous demande d'ajouter (si ce n'est déjà fait) une méthode **reset()** à la classe **PhotoShop** qui permettra de supprimer toutes les images de la bibliothèque, remettre **numCourant** à 1 et remettre les couleurs **couleurTrue** et **couleurFalse** de **ImageB** à BLANC et à NOIR respectivement. L'item de menu « Reset »



permettra de faire ce « reset ». Pour cela, on vous demande de compléter la méthode **void on_actionReset_triggered()**.

Etape 15 : BONUS (non obligatoire donc) : traitements d'images couleurs

Dans la version actuelle de l'application, il n'y aucune opération de traitements d'images couleurs… On pourrait imaginer de compléter la classe **ImageRGB** des méthodes suivantes :

- **ImageNG getRouge() const** : qui retournerait une **ImageNG** dont chaque valeur de niveau de gris correspondrait à la composante **rouge** des pixels correspondants.
- **ImageNG getVert() const** : qui retournerait une **ImageNG** dont chaque valeur de niveau de gris correspondrait à la composante **verte** des pixels correspondants.
- **ImageNG getBleu() const** : qui retournerait une **ImageNG** dont chaque valeur de niveau de gris correspondrait à la composante **bleue** des pixels correspondants.
- **void setRGB(const ImageNG &r, const ImageNG &g, const ImageNG& b)** : qui permettrait de mettre à jour les couleurs des pixels de l'image en les remplaçant par les composantes correspondantes des images r, g et b.

Ces méthodes permettraient donc de décomposer une **ImageRGB** en **3 ImageNG** qui pourraient alors être traitées séparément par les méthodes déjà développées dans la classe **Traitements**. On pourrait alors imaginer de réaliser le floutage d'une **ImageRGB** en floutant séparément ses 3 composantes de couleurs, puis en les « rassemblant » avec la méthode setRGB. Il en va de même pour d'autres traitements comme le filtre médian, l'augmentation/diminution de luminosité. La classe **Traitements** pourrait alors être complétée de traitements pour images couleurs. Ensuite, l'interface graphique pourrait être mise à jour afin d'en tenir compte.

Bon travail 😊 !