

Laboratoire de Programmation en C++

**2^{ème} informatique et systèmes :
option(s) industrielle et réseaux (1^{er} quadrimestre)
et 2^{ème} informatique de gestion (1^{er} quadrimestre)**

Année académique 2019-2020

« *Mini-PhotoShop* » en C++

**Anne Léonard
Denys Mercenier
Patrick Quettier
Claude Vilvens
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) 2^{ème} Bach. en informatique de Gestion : « 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) 2^{ème} Bach. en 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 2020 (sur base d'une liste de points de théorie fournis en novembre et à préparer) et coté sur 20
- ♦ laboratoire (cet énoncé) : une évaluation globale en janvier, accompagné de « check-points » réguliers pendant tout le quadrimestre. Cette évaluation fournit une note de laboratoire sur 20
- ♦ note finale : **moyenne arithmétique de la note de théorie (50%) et de la note de laboratoire (50%).**

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

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

2) En 2^{ème} 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 via leur centre de ressources

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.



Différentes entités vont donc apparaître : les notions d'images, de pixels, 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 de la globalité ou d'une partie de l'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 Unix 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 UNIX. La machine de développement sera Sunray2 (10.59.28.2 sur le réseau de l'école). Néanmoins, une machine virtuelle (VMWare) possédant exactement la même configuration que celle de Sunray2 sera mise à la disposition des

étudiants lors des premières séances de laboratoire. Mais attention, **seul le code compilable sur Sunray2 sera pris en compte !!!**

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.

De plus, pour mettre au point vos classes, vous disposerez de deux programmes de test appelés **mTestCopie.cpp** et **mTestEgal.cpp** qui vous permettront de valider les constructeurs de copie et les opérateurs égal de vos classes.

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 elle-même.

5. Planning et contenu des évaluations

a) Evaluation 1 (continue) :

Le développement de l'application, depuis la création des briques de base jusqu'à la réalisation du main 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.

Afin de **suivre votre évolution** (et surveiller celle des autres), vous disposerez d'une application fournie au laboratoire qui vous permettra de savoir où vous en êtes et de connaître les différents commentaires laissés par votre professeur de laboratoire.

b) Evaluation 2 (examen de janvier 2020) :

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 Unix de l'école, selon les modalités fixées par le professeur de laboratoire.

Plan des étapes à réaliser

Etape	Thème	Page
1	Une première classe	6
2	Associations entre classes : agrégation + Variables statiques	6
3	Mise en place de la matrice de pixels Utilisation de méthodes statiques, agrégation par utilisation	7
4	Surcharges des opérateurs	11
5	Associations de classes : héritage et virtualité	14
6	Les exceptions	17
7	BONUS (non obligatoire donc) : Un premier template : la classe Matrice	18
8	Première utilisation des flux	19
9	Les containers : une hiérarchie de templates	20
10	Traitements d'images simples : Création de méthodes statiques	22
11	Mise en place de l'application : les classes Moteur et UI	25
12	Sauvegarde de l'état de l'application : un fichier binaire	31
13	Importation d'images à partir d'un fichier csv : un fichier texte	31
14	Création d'un fichier de traces : un second fichier texte	32
15	BONUS (non obligatoire donc) : traitements d'images couleurs	33

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 la classe **string** et 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 chaîne 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. Les variables de type chaîne de caractères seront donc des char*. **Pour des raisons purement pédagogiques, le type string (de la STL) NE pourra PAS être utilisé du tout dans TOUT ce dossier de C++.** Vous aurez l'occasion d'utiliser la classe string dans votre apprentissage du C# et du Java.

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,

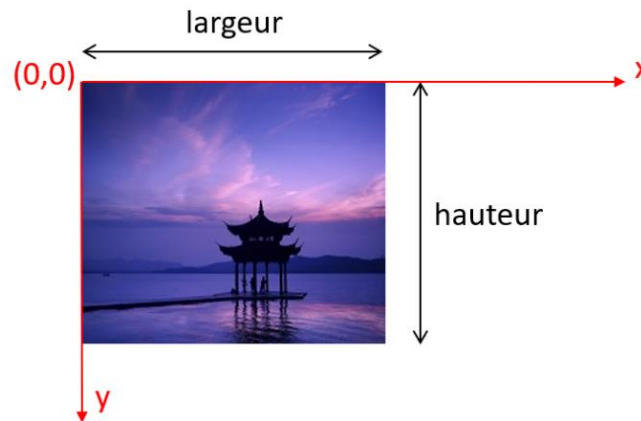
- Vous ne pouvez pas toucher aux prototypes des constructeurs existants ! Vous devez simplement vous arranger pour que les images créées par ces constructeurs aient la dimension par défaut **Dimension::VGA**.
- Vous devez ajouter un nouveau constructeur d'initialisation complet tenant compte de la dimension.
- 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 : allocation dynamique

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 :



La taille d'une image devra être gérée de manière dynamique. Pour cela, on demande d'ajouter à la classe **ImageNG** une variable **matrice** de type **int****. Pour l'allocation d'une telle matrice, les normes actuelles du C++ vous imposent de coder

```
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.
- La taille du tableau **matrice** est intimement liée à la variable **dimension** de la classe **ImageNG**. Une modification de l'une ne pourra pas se faire sans l'autre. Dès lors, c'est dans la méthode **setDimension** que vous devrez (dés)allouer la matrice en fonction du paramètre reçu par la méthode. N'oubliez pas de mettre à jour si nécessaire les constructeurs et destructeur de votre classe **ImageNG** !
- 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 de deux classes **WindowSDL** et **WindowSDLImage**.

WindowSDL gère une fenêtre graphique du système d'exploitation. Elle contient les méthodes statiques suivantes :

- **void open(int w,int h)** : qui permet d'ouvrir une fenêtre graphique de **w** pixels de large et de **h** pixels de haut. Sachez qu'**une seule fenêtre graphique ne peut être ouverte à la fois.**
- **void setBackground(int r,int v,int b)** : qui permet de dessiner un fond uniforme de couleur (r,v,b).
- **void close()** : qui permet de fermer la fenêtre.
- différentes méthodes permettant de dessiner des pixels, lignes, rectangles pleins ou non.
- **void drawImage(WindowSDLImage image, int x,int y)** qui permet d'afficher une image à l'emplacement (x,y) de la fenêtre graphique.
- La méthode bloquante **waitClick()** qui attend que l'utilisateur clique sur la fenêtre. Une fois qu'il a cliqué, la méthode retourne un objet de classe **WindowSDLclick** qui contient les coordonnées (x,y) du pixel sur lequel l'utilisateur a cliqué. **Si l'utilisateur clique sur la croix de la fenêtre, la méthode retourne (-1,-1).**

WindowSDLImage est une classe qui sert d'intermédiaire entre votre programme (et donc vos classes) et les fichiers bitmaps (BMP). Elle gère donc en mémoire une image BMP (qui peut provenir par exemple d'un fichier bitmap). Elle possède les méthodes suivantes :

- **void importFromBMP(const char* fichier)** qui permet de charger en mémoire (quelque part dans ses variables membres → on ne veut pas savoir → vive l'encapsulation 😊) l'ensemble des informations (taille, pixels, ...) d'une image BMP à partir d'un fichier dont on passe le nom en paramètre.
- **void exportToBMP(const char* fichier)** qui permet de créer un fichier BMP dont on fournit le nom en paramètre.
- **int getWidth()** et **int getHeight()** qui retournent respectivement la largeur et la hauteur de l'image BMP.
- **void setPixel(...)** et **void getPixel(...)** qui permettent de modifier/récupérer les composantes RGB d'un pixel.

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

1. ouvre la fenêtre graphique à la dimension de l'image (variable membre **dimension**),

2. crée un objet WindowSDLImage temporaire,
3. remplit cet objet avec les valeurs des niveaux de gris de la variable membre **matrice** (utilisation de setPixel de WindowSDLImage),
4. dessine cet objet dans la fenêtre graphique (utilisation de WindowSDL::drawImage),
5. attend un clic de l'utilisateur (utilisation de WindowSDL::waitClick()),
6. ferme la fenêtre graphique.

Le souci est qu'actuellement notre classe **ImageNG** ne peut gérer que des images en niveaux de gris tandis que WindowSDLImage nécessite de connaître les composantes RGB de la couleur. Il suffit pour cela de mettre les trois composantes RGB égales à la valeur du niveau de gris.

Remarque théorique : La méthode **Dessine()** de la classe **ImageNG** utilise l'**agrégation par utilisation**. En effet, elle instancie un objet WindowSDLImage, l'utilise (pour pouvoir dessiner dans la fenêtre graphique), puis détruit cet objet après usage.

c) Importation/Exportation de et vers des fichiers bitmap

Nous allons à présent modifier la classe **ImageNG** afin qu'elle puisse importer/exporter une image à partir d'un/vers un fichier BMP. On vous demande tout d'abord d'ajouter à la classe **ImageNG** la méthode **void importFromBMP(const char* fichier)** qui

1. crée un objet WindowSDLImage temporaire grâce au nom de fichier reçu en paramètre. Un constructeur de WindowSDLImage est prévu à cet effet,
2. alloue la **matrice** grâce aux dimensions de l'image BMP lue sur disque. Si tout a été fait correctement jusque là, la méthode **setDimension** de votre classe **ImageNG** fera le travail.
3. remplit la **matrice** avec les données stockées dans l'objet temporaire (utilisation de getPixel de WindowSDLImage),

A nouveau, actuellement, notre classe **ImageNG** ne peut gérer que des images en niveaux de gris tandis que WindowSDLImage fournit les composantes RGB de la couleur des pixels. Une conversion de la couleur vers un niveau de gris est donc nécessaire. Il suffit simplement de stocker dans la matrice la luminance de chaque pixel. La luminance d'une couleur est simplement la moyenne de ses 3 composantes RGB : $\text{luminance} = (\text{rouge} + \text{vert} + \text{bleu}) / 3$.

Ensuite, on vous demande d'ajouter à la classe **ImageNG** la méthode **void exportToBMP(const char* fichier)** qui

1. crée un objet WindowSDLImage temporaire,
2. remplit cet objet avec les valeurs des niveaux de gris de la variable membre **matrice** (utilisation de setPixel de WindowSDLImage),
3. crée sur disque le fichier BMP dont le nom est fourni en paramètre (utilisation exportToBMP de WindowSDLImage).

Même remarque de précédemment : WindowSDLImage nécessite de connaître les composantes RGB de la couleur. Il suffira pour cela de mettre les trois composantes RGB égales à la valeur du niveau de gris.

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. Dans ce but, une nouvelle classe (**PixelNG**) sera également développée.

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

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

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 la compilation et 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");  
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 la compilation et 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) La classe PixelNG (Essai9())

Afin de pouvoir modifier les pixels d'une image à l'aide d'opérateurs, on vous demande créer la classe **PixelNG** qui possède les variables membres suivantes :

- **x,y** : deux entiers qui représentent la position du pixel dans l'image.
- **valeur** : un entier qui représente la valeur du niveau de gris.

On vous demande de créer les accesseurs et les constructeurs classiques, ainsi que l'opérateur << de cette classe.

g) Opérateurs (ImageNG + PixelNG) et (PixelNG + ImageNG) de la classe Image (Essai10())

Additionner un pixel à une image revient simplement à remplacer le pixel correspondant de l'image par la nouvelle valeur. En d'autres termes, l'opérateur + revient à « dessiner le nouveau pixel par-dessus l'ancien ». On vous demande donc de surcharger l'opérateur + de la classe **ImageNG** permettant d'exécuter un code du genre :

```
ImageNG i1("boat.bmp"), i2, i3;  
i2 = i1 + PixelNG(100,120,200); // i2 correspond à l'image i1 mais dans laquelle  
                                // le pixel (100,120) a une valeur de 200  
                                // i1 doit rester inchangée !  
i3 = PixelNG(20,60,100) + i2;
```

h) Surcharge des opérateurs de comparaison (== et !=) de la classe Dimension (Essai11())

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.

i) Surcharge de l'opérateur (ImageNG - ImageNG) de la classe ImageNG (Essai12())

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.

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

Comparer deux images peut se faire selon différents critères. Par exemple,

- selon la luminance : une image sera < (> ou ==) à une autre si la valeur de sa luminance (valeur moyenne des pixels de l'image) est < (> ou ==) à la luminance de l'autre image.
- selon une comparaison 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.

Les deux critères de comparaison devront être programmés. Pour cela, on demande d'ajouter à la classe **ImageNG**

- **deux variables membres statiques publiques constantes** de type **int** : **LUMINANCE** et **PIXEL_A_PIXEL** représentant les deux modes possibles de comparaison.
- **une variable membre statique privée** de type **int** appelée **comparaison** et qui définira le mode de comparaison de toutes les images instanciant la classe **Image**.
- **deux méthodes statiques publiques void** **setComparaison(int c)** et **int getComparaison()** permettant de modifier proprement (→ comparaison ne peut prendre qu'une des deux valeurs LUMINANCE ou PIXEL_A_PIXEL) et d'obtenir la variable comparaison.

On vous demande ensuite de programmer les opérateurs <, > et == de la classe **ImageNG** qui comparent deux images selon le mode de comparaison en cours :

```
ImageNG i1, i2;
...
ImageNG::setComparaison(ImageNG::LUMINANCE);
if (i1 < i2) ...
if (i1 > i2) ... // comparaison selon la luminance
if (i1 == i2) ...
ImageNG::setComparaison(ImageNG::PIXEL_A_PIXEL);
if (i1 < i2) ...
if (i1 > i2) ... // comparaison pixel à pixel
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 les petites hiérarchies 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***) et **dimension** (**Dimension**)
- les accesseurs **getXXX/setXXX** pour ces trois variables membres. Attention que la méthode **setDimension** ne fera que modifier la variable membre dimension (l'allocation de la matrice de pixels se fera dans les classes héritées).
- Les **méthodes virtuelles pures** **void Affiche()**, **void Dessine()** et **void exportToBMP(const char* fichier)**

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 type **int**** représentant la matrice de pixels (valeurs de niveaux de gris)
- une méthode **setDimension** réalisant la (dés)allocation dynamique de la matrice, comme cela était fait précédemment.
- toutes les méthodes, opérateurs, variables membres statiques 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).

Une remarque s'impose tout de même concernant la classe **PixelNG**. On se rend bien compte qu'il va y avoir des classes dédiées aux pixels des images binaires et couleurs. Donc, dans le même ordre d'idée que précédemment, on demande de créer la classe **Pixel** qui comporte

- Les variables membres **x** et **y** (**int**)
- Les accesseurs classiques getXXX/setXXX pour ces variables membres.
- Les 3 constructeurs classiques.

Ensuite, on vous demande de modifier la classe **PixelNG** de telle sorte qu'elle

- hérite de la classe **Pixel**,
- possède en plus la variable membre **valeur** (**int**) représentant le niveau de gris du pixel,
- possède les constructeurs et accesseurs classiques, ainsi qu'un opérateur <<.

Ainsi tout se passe en totale transparence pour l'utilisateur de vos classes qui ne se rend compte de rien... C'est à nouveau tout le principe et l'intérêt de l'encapsulation.

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 type **Couleur**** représentant la matrice de pixels de couleur,
- la méthode **setDimension()** permettant de (dés)allouer la matrice de pixels de l'image comme cela a été fait avec les int pour les images en niveaux de gris.
- 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 **void importFromBMP(const char* fichier)** et **void exportToBMP(const char* fichier)**,
- les opérateurs =, << classiques
- les opérateurs permettant de réaliser les opérations (**ImageRGB + PixelRGB**) et (**PixelRGB + ImageRGB**). **PixelRGB** est une classe héritée de **Pixel** et qui dispose en plus d'une variable membre **valeur** de type **Couleur**, et que vous devez donc programmer (constructeurs et accesseurs classiques, opérateur <<). Cette classe est donc l'équivalent de **PixelNG** mais pour les couleurs.

Pour tester **ImageRGB** et **PixelRGB** : Test5.cpp, Essai2() à Essai5().

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 type **bool**** représentant la matrice de pixels binaires,
- la méthode **setDimension()** permettant de (dés)allouer la matrice de pixels de l'image comme cela a été fait avec les int et Couleur précédemment.
- 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 BMP) 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 exportToBMP(const char* fichier)**. Remarquez qu'une importation à partir d'un fichier BMP n'est pas possible dans le cas des images binaires.
- les opérateurs = et <<
- les opérateurs permettant de réaliser les opérations (**ImageB + PixelB**) et (**PixelB + ImageB**). **PixelB** est une classe héritée de **Pixel** et qui dispose en plus d'une variable membre **valeur** de type **bool**, et que vous devez donc programmer (constructeurs et accesseurs classiques, opérateur <<). Cette classe est donc l'équivalent de **PixelNG** et **PixelRGB** mais pour les valeurs binaires.

Pour tester **ImageB** et **PixelB** : Test5.cpp, Essai6() à Essai8().

d) Mise en évidence de la virtualité et du down-casting : Essai9() et Essai10()

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++.

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 :

- **BaseException** : Cette classe contiendra une seule variable membre du type **chaîne de caractères (char *)**. 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 **BaseException** 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 **BaseException** 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.
 - créer/modifier un **PixelNG** avec un niveau de gris non compris 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 **BaseException** 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 **BaseException** 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)
 - créer/modifier un **Pixel** avec des coordonnées non valides (c'est-à-dire négatives).
 - comparer des images en niveau de gris **ImageNG** qui n'ont pas les mêmes dimensions (comparaison PIXEL_A_PIXEL, 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) : **BONUS (non obligatoire donc) : Un premier template : la classe** **Matrice**

Les classes d'images développées jusqu'ici embarquent toutes les trois le même code d'allocation dynamique, gestion et désallocation d'une matrice de pixels. Ce code est quasi identique dans les trois classes à la différence que le type de données est différent. Il serait donc intéressant d'intégrer ce code dans une classe template dédiée.

a) La classe Matrice (Test7.cpp)

On vous demande donc de programmer la classe **Matrice template** contenant comme variable membre le pointeur, de type **T****, vers le premier élément de la matrice. Elle aura donc la structure de base suivante :

```
template<class T> class Matrice
{
    private :
        T** tab;
    ...
}
```

La classe **Matrice** devra disposer de :

- deux variables membres **largeur** et **hauteur**, de type **int**, précisant la taille de la matrice.
- un **constructeur par défaut** allouant la matrice aux dimensions 3x3.
- un constructeur d'initialisation **Matrice(int l,int h)** allouant la matrice aux dimensions voulues
- un constructeur d'initialisation **Matrice(int l,int h,const T& val)** permettant d'allouer la matrice aux dimensions voulues et d'initialiser tous ses éléments à la valeur val.
- Un **constructeur de copie**.
- Un **destructeur** permettant de libérer correctement la mémoire.
- Les méthodes **void setValeur(int x,int y,const T& val)** et **T getValeur(int x,int y) const** permettant de modifier/récupérer les éléments qui se trouvent dans la matrice, **x** étant l'indice de colonne et **y** l'indice de ligne.
- Les méthodes **getLargeur()** et **getHauteur()** retournant les dimensions de la matrice
- La méthode **Affiche()** permettant d'afficher chaque élément de celle-ci sous forme d'un tableau dans la console.
- L'opérateur d'affectation = classique.
- Les opérateurs **T operator()(int x,int y) const** et **T& operator()(int x,int y)** permettant de manipuler la matrice facilement. Exemple :

```
Matrice<int> m(5,7) ; // appel au constructeur
m(2,4) = 6 ; // appel à l'opérateur
```

Le premier opérateur sert à lire sans modifier la matrice tandis que le second permet de la modifier.

b) Refactoring des classes ImageNG, ImageRGB et ImageB

Il s'agit simplement de mettre à jour ces trois classes en remplaçant les variables **int****, **Couleur**** et **bool**** par des instances des classes **Matrice<int>**, **Matrice<Couleur>** et **Matrice<bool>**. Normalement, si vos développements ont tenu compte des avantages de l'encapsulation, seules les méthodes **setPixel** et **getPixel** devront être modifiées (sinon vous disposez des opérateurs () de la classe **Matrice**).

Pour tester les mises à jour de vos classes, il suffit de tester à nouveau tous les jeux de tests développés jusque maintenant.

Remarquez que **la variable membre n'est plus un pointeur mais bien un objet** ! Veillez donc à en tenir compte dans vos mises à jour.

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 demande 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, dimensions et matrice de pixels) et cela champ par champ. Le fichier obtenu sera un fichier **binaire** (utilisation des méthodes **write** et **read**).
- ♦ **Load(istream & 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(istream & 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.

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

Etape 9 (Test9.cpp) :

Les containers : une hiérarchie de 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 toutes les images, ou encore un ensemble de valeurs de pixels en vue de leur traitement (voir plus loin). Nous allons ici mettre en place une base pour nos containers. Ceux-ci seront construits via une hiérarchie de classes templates.

b) Le container typique : la liste

Le cœur de notre hiérarchie va être une liste chaînée dynamique. 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 abstraite **ListeBase 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 ListeBase
{
    protected :
        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 **ListeBase** 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'afficher chaque élément de celle-ci.
- La **méthode virtuelle pure void insere(const T & val)** qui permettra, une fois redéfinie dans une classe héritée, d'insérer un nouvel élément dans la liste, à un endroit dépendant du genre de liste héritée (simple liste, pile, file, liste triée, ...).
- 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.

c) Une première classe dérivée : La liste simple (Essai1() et Essai2())

Nous disposons à présent de la classe de base de notre hiérarchie. La prochaine étape consiste à créer la **classe template Liste** qui hérite de la classe ListeBase et qui redéfinit la méthode insere de telle sorte que **l'élément ajouté à la liste soit inséré à la fin de celle-ci**.

Dans un premier temps, vous testerez votre classe Liste 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.

d) La liste triée (Essai3() et Essai4())

On vous demande à présent de programmer la **classe template ListeTrie** qui hérite de la classe ListeBase et qui redéfinit la méthode insere de telle sorte que l'élément ajouté à la liste soit inséré au bon endroit dans la liste, c'est-à-dire en respectant l'ordre défini par les opérateurs de comparaison de la classe template.

Dans un premier temps, vous testerez votre classe ListeTrie avec des **entiers**, puis ensuite avec des objets de la classe **ImageNG**. Celles-ci devront être triés par ordre croissant de luminance.

e) Parcourir et modifier une liste : l'itérateur de liste (Essai5() à Essai8())

Dans l'état actuel des choses, nous pouvons ajouter des éléments à une liste ou à une liste triée mais nous n'avons aucun moyen de parcourir cette liste, élément par élément, afin de les afficher ou de faire une recherche. La notion d'itérateur va nous permettre de réaliser ces opérations.

On vous demande donc de créer la classe **Iterateur** qui sera un **itérateur** de la classe **ListeBase** (elle permettra donc de parcourir tout objet instanciant la classe **Liste** ou **ListeTrie**), 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.
- **T remove()** qui supprime de la liste et retourne l'élément pointé par l'itérateur.
- **Opérateur ++** qui déplace l'itérateur vers la droite.
- **Opérateur de casting ()** qui retourne (par valeur) l'élément pointé par l'itérateur.
- **Opérateur &** qui retourne (par référence) l'élément pointé par l'itérateur.

L'application finale fera un usage abondant des containers. On vous demande donc d'utiliser la classe Iterateur afin de vous faciliter l'accès aux containers. Son usage sera vérifié lors de l'évaluation finale.

Etape 10 (Test10.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 classer 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

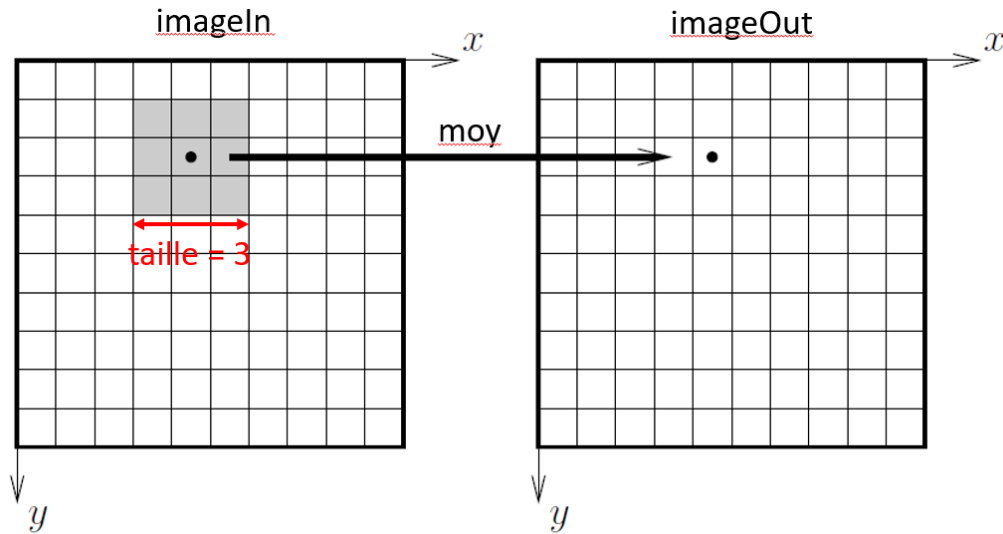
ImageB Seuillage(const ImageNG& imageIn, int seuil) ;

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**.

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 $3 \times 3 = 9$ pixels situés autour du pixel traité •. La taille du filtre sera toujours un nombre impair, ce qui correspond à moyenner les $3 \times 3 = 9$, $5 \times 5 = 25$, $7 \times 7 = 49$, ... pixels situés autour du pixel traité.

L'effet de ce filtre est de flouter la zone où les pixels seront traités par ce processus. On vous demande donc de programmer la méthode statique

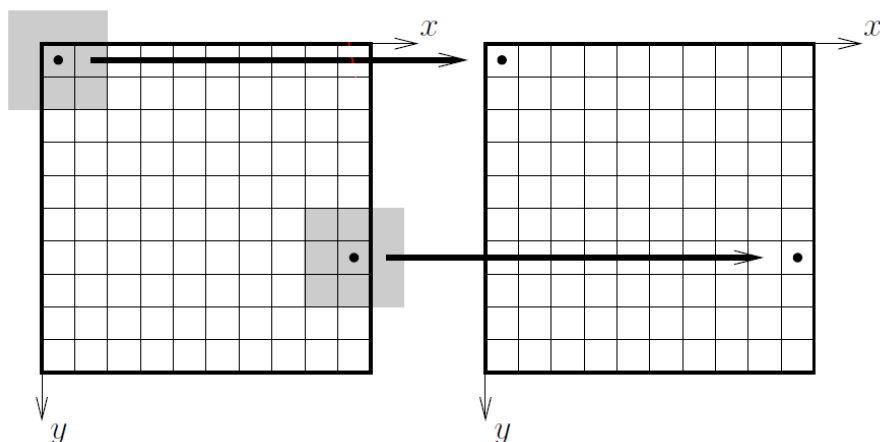
ImageNG FiltreMoyenneur(const ImageNG& imageIn, int taille, Pixel p1=Pixel(), Pixel p2=Pixel()) ;

L'image de sortie est une ImageNG de même taille que imageIn. Le paramètre taille correspond à la taille du filtre, et les pixels p1 et p2 délimitent la zone rectangulaire de l'image où le filtrage doit être appliqué.

Remarquez que la fonction présente **deux paramètres par défaut**. Si ces paramètres sont utilisés par défaut, tous les pixels de l'imageIn doivent être filtrés, l'image est donc filtrée dans son entièreté.

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 d'utiliser une `ListeTriee<int>` temporaire dans laquelle vous insérerez les valeurs des $3 \times 3 = 9$, $5 \times 5 = 25$, $7 \times 7 = 49$, ... pixels situés autour du pixel traité, et d'en extraire la valeur centrale. 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, et on remarquera que tous les pixels de l'image seront traités.

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

Les opérations d'érosion et dilatation sur une ImageNG sont à nouveau similaires 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.

e) Négatif d'une ImageNG (Essai7())

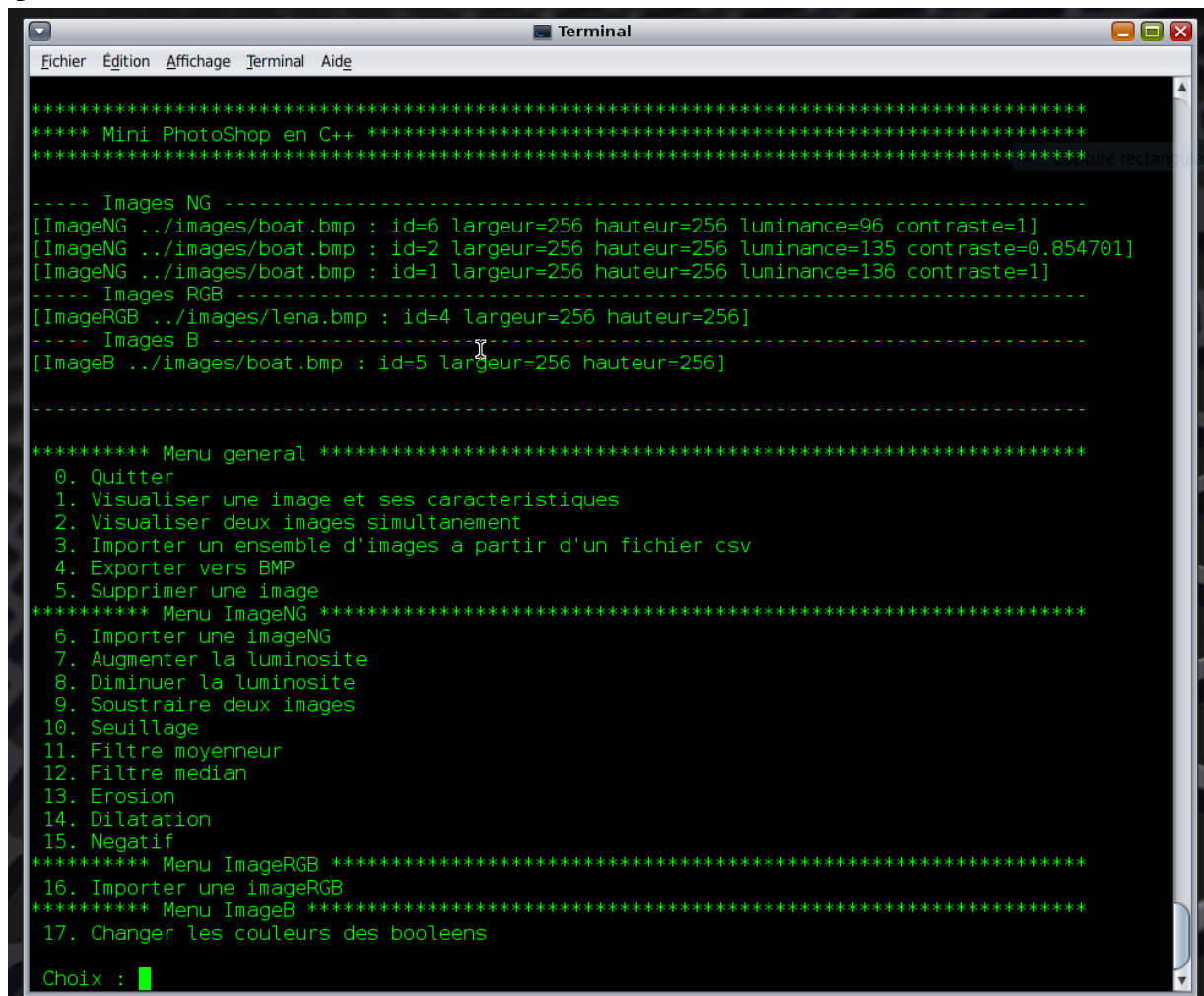
Le négatif d'une image en niveaux de gris consiste à remplacer chaque valeur **X** de pixel par son « complément », c'est-à-dire par la valeur **(255-X)**. Tous les pixels de mêmes niveaux de gris sont donc remplacés par la même valeur indépendamment de leurs voisins. Cette technique entre dans une famille 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 Négatif(const ImageNG& imageIn);

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

Etape 11 : Mise en place de l'application : les classes Moteur et UI

Nous allons à présent mettre en place l'application proprement dite. Celle-ci, une fois lancée, devra proposer le menu suivant



```
***** Mini PhotoShop en C++ *****
----- Images NG -----
[ImageNG ../images/boat.bmp : id=6 largeur=256 hauteur=256 luminance=96 contraste=1]
[ImageNG ../images/boat.bmp : id=2 largeur=256 hauteur=256 luminance=135 contraste=0.854701]
[ImageNG ../images/boat.bmp : id=1 largeur=256 hauteur=256 luminance=136 contraste=1]
----- Images RGB -----
[ImageRGB ../images/lena.bmp : id=4 largeur=256 hauteur=256]
----- Images B -----
[ImageB ../images/boat.bmp : id=5 largeur=256 hauteur=256]

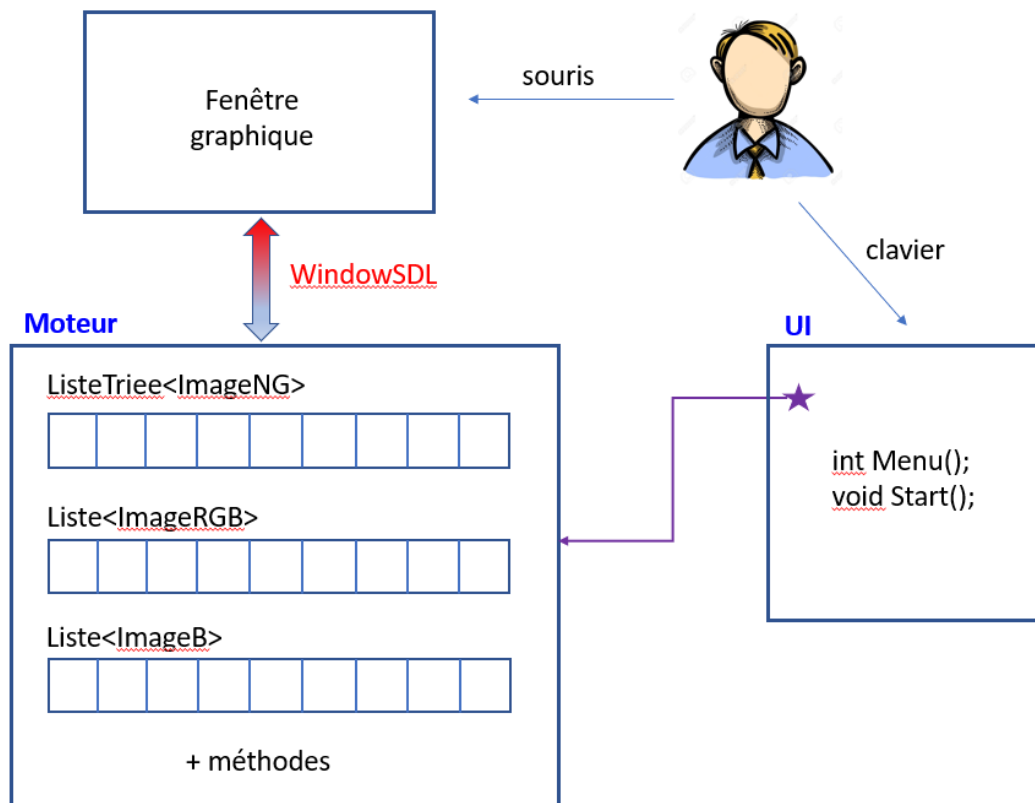
***** Menu general *****
 0. Quitter
 1. Visualiser une image et ses caracteristiques
 2. Visualiser deux images simultanement
 3. Importer un ensemble d'images a partir d'un fichier csv
 4. Exporter vers BMP
 5. Supprimer une image
***** Menu ImageNG *****
 6. Importer une imageNG
 7. Augmenter la luminosite
 8. Diminuer la luminosite
 9. Soustraire deux images
10. Seuillage
11. Filtre moyennneur
12. Filtre median
13. Erosion
14. Dilatation
15. Négatif
***** Menu ImageRGB *****
16. Importer une imageRGB
***** Menu ImageB *****
17. Changer les couleurs des booleans

Choix : █
```

A chaque (re-)affichage de celui-ci, la liste des images présentes en mémoire apparaîtra en haut.
L'application sera architecturée à l'aide de deux classes :

- la classe **Moteur** qui contiendra l'ensemble des images en mémoire sous la forme de variables membres (conteneurs). Cette classe contiendra toute la logique métier : l'appel des méthodes de traitements d'images, gestion des identifiants (uniques) des images, manipulation de la fenêtre graphique à l'aide de la librairie WindowSDL.
- la classe **UI** (UI pour « User Interface ») qui gèrera et affichera le menu tout en manipulant une instance de la classe **Moteur**, dont elle possède un pointeur.

Voici un schéma décrivant la situation :



a) La classe Moteur

On vous demande de créer la classe **Moteur** dont la structure est la suivante :

```
class Moteur
{
private :
    ListeTrie<ImageNG> imagesNG ;
    Liste<ImageRGB>     imagesRGB ;
    Liste<ImageG>       imagesB ;
    ...
public :
    ...
} ;
```

Cette classe dispose donc des variables membres suivantes :

- **imagesNG** : qui contient l'ensemble des images en niveaux de gris triées par ordre croissant de luminance.
- **imagesRGB** : qui contient l'ensemble des images RGB de l'application.
- **imagesB** : qui contient l'ensemble des images binaires de l'application.

D'autres variables membres pourraient être ajoutées en fonction de vos besoins.

On vous demande de programmer les **méthodes privées** (donc uniquement utilisables par les méthodes publiques de la classe) suivantes :

- **int Insere(Image *pImage)** : qui permet d'insérer, dans le bon conteneur, une image (quelque soit son type) dont on passe l'adresse en paramètre à la fonction. Elle attribuera (et retournera) un identifiant unique à l'image insérée. Pour ce faire, on vous demande d'utiliser le **dynamic-cast du C++**.
- **ImageNG GetImageNG(int id)** : qui retourne l'ImageNG d'identifiant id présente dans le conteneur imagesNG. Pour ce faire, on vous demande d'utiliser un itérateur.
- **Image* GetImage(int id)** : qui retourne l'adresse de l'image dont l'identifiant id est passé en paramètre à la fonction. De nouveau, on vous demande d'utiliser des itérateurs.

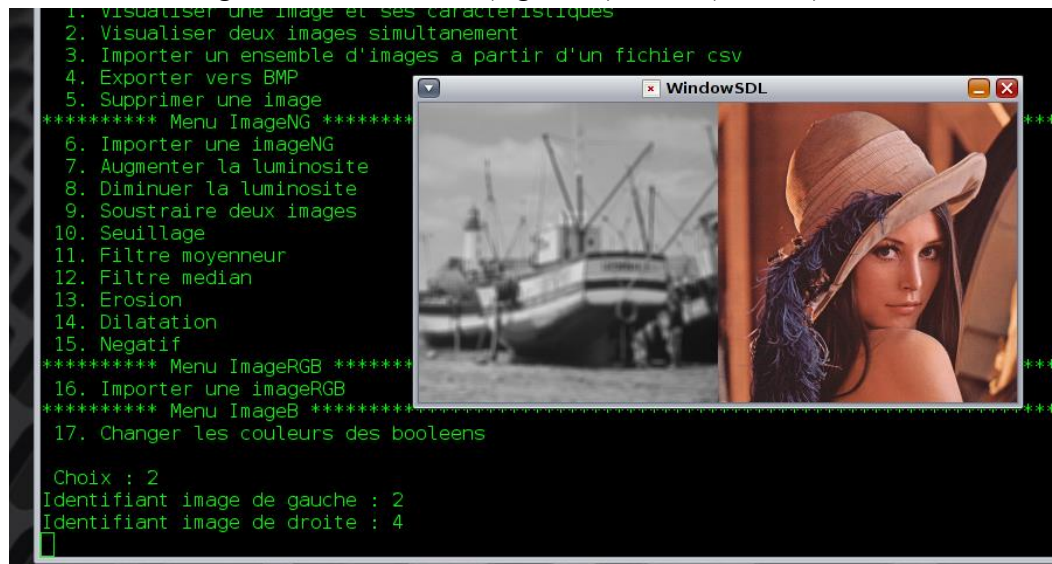
Que se passe-t-il si le paramètre id ne correspondant à aucune image présente dans un des conteneurs ? Dans ce cas, une exception du type **MoteurException** doit être lancée (celle-ci est simplement une classe dérivant de **BaseException** mais ne présentant pas de variable en plus ; vous devez donc également programmer cette classe).

On vous demande ensuite de programmer les méthodes publiques suivantes (ces méthodes appelleront donc les méthodes privées décrites ci-dessus en fonction de leurs besoins) :

- **void Affiche()** : qui permet d'afficher dans la console le contenu des 3 conteneurs. Cette méthode sera utilisée par la classe UI lorsque le menu devra être affiché.
- **void SupprimeImage(int id)** : qui permet de supprimer l'image dont l'identifiant est id.
- **int ImporteImageNG(const char * fichier)** : qui permet d'importer une image BMP, de créer une instance de la classe ImageNG, objet qui sera inséré dans le bon conteneur. La méthode doit retourner l'identifiant qui a été attribué à l'image.
- **int ImporteImageRGB(const char * fichier)** : qui permet d'importer une image BMP, de créer une instance de la classe ImageRGB, objet qui sera inséré dans le bon conteneur. La méthode doit retourner l'identifiant qui a été attribué à l'image.
- **void VisualiseImage(int id)** : qui permet d'afficher toutes les caractéristiques de l'image d'identifiant id dans la console et de la dessiner dans une fenêtre graphique (utilisation de la méthode Dessine() de la classe Image) :



- **void VisualiseImages(int id1, int id2)** : qui permet de dessiner simultanément et dans la même fenêtre les images d'identifiant id1 (à gauche) et id2 (à droite) :



Pour ce faire, vous devez ajouter à la classe **Image** la **méthode virtuelle pure void Dessine(int x,int y)** (méthode qui sera donc redéfinie dans chaque classe héritée). Cette méthode doit dessiner l'image en positionnant son coin supérieur gauche aux coordonnées (x,y) de la fenêtre graphique. Cette méthode Dessine ne doit ni ouvrir, ni fermer la fenêtre, ni attendre un clic de l'utilisateur. C'est à la méthode **VisualiseImages** de calculer la taille de la fenêtre, de l'ouvrir, d'appeler les méthodes Dessine(x,y) des deux images, d'attendre un clic utilisateur et de fermer la fenêtre.

- **void ExporterBMP(int id, const char* fichier)** : qui permet d'exporter et d'enregistrer au format BMP (dans le fichier) l'image d'identifiant id.
- **int AugmenteLuminosite(int id, int val)** : qui permet de créer une nouvelle image en augmentant la luminosité de l'image id d'une valeur val. Cette méthode ne sera utilisable que pour une ImageNG et utilisera donc l'opérateur + de cette classe. Elle insérera l'image obtenue dans le bon conteneur et retournera son identifiant.
- **int DiminueLuminosite(int id, int val)** : qui permet de créer une nouvelle image en diminuant la luminosité de l'image id d'une valeur val. Cette méthode ne sera utilisable que pour une ImageNG et utilisera donc l'opérateur - de cette classe. Elle insérera l'image obtenue dans le bon conteneur et retournera son identifiant.

- **int Soustraire(int id1, int id2)** : qui permet de créer une nouvelle image en calculant la différence entre l'image id1 et l'image id2. Cette méthode ne sera utilisable que pour des ImageNG et utilisera donc l'opérateur - de cette classe. Elle insérera l'image obtenue dans le bon conteneur et retournera son identifiant.

Les 6 méthodes qui suivent correspondant aux 6 méthodes statiques de la classe **Traitements** :

- **int Seuillage(int id, int seuil)**
- **int Negatif(int id)**
- **int FiltreMedian(int id, int tailleFiltre)**
- **int FiltreMoyenieur(int id, int tailleFiltre)** → l'entièrement de l'image doit être traitée
- **int Dilatation(int id, int tailleFiltre)**
- **int Erosion(int id, int tailleFiltre)**

A chaque fois, l'image obtenue sera insérée dans le bon conteneur et son identifiant sera retourné par la méthode.

La dernière méthode va permettre de flouter une zone de l'image sélectionnée par l'utilisateur. Elle ne sera utilisable que pour une ImageNG. Il s'agit de la méthode **int FiltreMoyenneurSelection(int id, int tailleFiltre)**. Cette méthode doit

1. ouvrir la fenêtre graphique et y dessiner l'image id (à l'aide de sa méthode Dessine(x,y))
2. attendre un premier clic de l'utilisateur et créer un objet Pixel p1
3. attendre un second clic de l'utilisateur et créer un objet Pixel p2
4. appeler la méthode **FiltreMoyenneur** de la classe **Traitement** afin de flouter la zone délimitée par les pixels p1 et p2
5. dessiner l'image obtenue dans la fenêtre (à la place de l'image originale)
6. recommencer au point (2) jusqu'au moment où l'utilisateur cliquera sur la croix de la fenêtre graphique → cela va permettre à l'utilisateur de flouter plusieurs zones s'il le souhaite.
7. insérer l'image obtenue dans le bon conteneur et retourner son identifiant.

Notez que, pour toutes les méthodes concernées, dès que le paramètre id n'est pas valide ou que la taille du filtre n'est pas adaptée, une exception **MoteurException** doit être lancée.

b) La classe UI

On vous demande de créer la classe **UI** dont la structure est la suivante :

```
class UI
{
private :
    Moteur * moteur ;
public :
    void Menu() ;
    void Start() ; ...
} ;
```

Outres ses indispensables constructeurs, cette classe dispose donc de :

- un pointeur **moteur** vers une instance de la classe **Moteur** qu'elle pourra contrôler.
- une méthode **Menu()** qui, après avoir appelé la méthode **Affiche()** de moteur, affiche le menu dans la console.
- une méthode **Start()** qui prend le contrôle du processus tout au long de son exécution. Cette méthode contient une boucle dans laquelle
 - elle affiche le menu (appel de **Menu()**),
 - elle demande le choix de l'utilisateur,
 - en fonction du choix réalisé, elle demande d'autres paramètres (id d'une image, nom de fichier, taille du filtre, seuil, ...) puis appelle la méthode de moteur correspondant à la demande de l'utilisateur. Comme vous l'aurez remarqué, chaque item du menu correspond à une méthode bien précise de Moteur.
 - elle affiche l'id et dessine l'image obtenue par le traitement,
 - elle remonte dans sa boucle afin de réafficher le menu.

Dans le cas du filtre moyennneur, il faudra demander à l'utilisateur s'il veut filtrer l'image entière ou une zone de l'image.

N'oubliez pas de gérer **MoteurException**... !!!

c) Mise en route de la mécanique : le main()

Reste maintenant à tout instancier et à lancer la boucle principale de votre programme. Votre main devrait ressembler à ceci :

```
int main()
{
    Moteur moteur ;
    UI ui(&moteur) ; // le constructeur fait le lien entre l'UI et le moteur

    ui.Start() ;

    return 0 ;
}
```

Etape 12 : Sauvegarde de l'état de l'application : un fichier binaire

Lorsque l'on quitte l'application, l'application doit sauvegarder son état (images contenues dans les conteneurs) dans un fichier binaire qui sera, par la suite, chargé au démarrage.

On vous demande donc de

- ajouter les méthodes **Save(ofstream &)** et **Load(ifstream &)** à la classe **Moteur** afin qu'elle (de)séréalise son état (au format binaire), c'est-à-dire le **contenu des 3 conteneurs d'images**. Pour ce faire, le plus simple est d'utiliser un itérateur et les méthodes **Save()** et **Load()** de vos classes images.
- faire en sorte que lorsque l'on quitte l'application (0 dans le menu), celle-ci enregistre son état dans le fichier binaire « **sauvegarde.dat** ».
- Au démarrage, l'application doit tenter d'ouvrir le fichier « **sauvegarde.dat** » afin de restaurer l'état du moteur. Si celui-ci n'existe pas, un moteur « vierge » doit être utilisé.

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

Jusque maintenant, l'item 3 du menu n'a pas encore été implémenté. Il serait en effet 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
RGB:../images/lena.bmp
...
NG:../images/house.bmp
...
RGB:../images/mandrill.bmp
```

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 deux informations :

1. le type d'image que l'on veut importer,
2. le nom du fichier BMP que l'on veut lire.

On vous demande donc d'ajouter à la classe **Moteur** la méthode **int ImporteCSV(const char* nomFichier)** qui doit

- ouvrir le fichier dont le nom est passé en paramètre. Si ce fichier n'existe pas, une **MoteurException** doit être lancée.
- lire le fichier ligne par ligne : en fonction du type d'image lu (« NG » ou « RGB »), elle doit créer un objet du bon type et l'insérer dans le conteneur correspondant.
- retourner le nombre d'images correctement importées.

Attention : Si une ligne du fichier csv correspond à un fichier BMP non valide, la lecture dans le fichier doit néanmoins continuer jusqu'au bout.

Vous devez ensuite modifier la classe **UI** pour tenir compte de cette nouvelle méthode.

Etape 14 : **Création d'un fichier de traces : un second fichier **texte****

Jusque maintenant, au fur et à mesure que l'on réalise des traitements et que l'on crée des nouvelles images, nous n'avons aucune trace de ce qui s'est passé. Il serait intéressant qu'au fur et à mesure que l'application est utilisée, un **fichier texte** soit alimenté avec les « traces » de ce qui est fait. Par exemple :

```
Ouverture du fichier csv images.csv pour importation...
Importation ImageNG a partir du fichier ../images/lena.bmp --> id = 1
Importation ImageRGB a partir du fichier ../images/lena.bmp --> id = 2
Importation ImageRGB a partir du fichier ../images/joconde.bmp --> id = 3
Importation ImageNG a partir du fichier ../images/house.bmp --> id = 4
Importation ImageRGB a partir du fichier ../images/bulles.bmp --> id = 5
Importation ImageRGB a partir du fichier ../images/mandrill.bmp --> id = 6
Importation ImageNG a partir du fichier ../images/boat.bmp --> id = 7
--> 7 images importees avec succes.
Image 2 exportee vers fichier BMP coucou.bmp
ImageRGB 5 supprimee.
Augmentation Luminosite de ImageNG 7 de +35 --> id = 8
Diminution Luminosite de ImageNG 8 de -50 --> id = 9
Seuillage de ImageNG 8 avec un seuil de 130 --> id = 10
Negatif de ImageNG 8 --> id = 11
Filtre median de ImageNG 9 taille filtre = 5 --> id = 12
Erosion de ImageNG 1 taille filtre = 5 --> id = 13
Dilatation de ImageNG 13 taille filtre = 5 --> id = 14
Soustraction de ImageNG 1 et 14 --> id = 15
Filtre moyennneur de ImageNG 9 taille filtre = 7 --> id = 16
Filtre moyennneur (Zone) de ImageNG 9 taille filtre = 11 --> id = 17
...
```


Pour ce faire, on vous demande d'ajouter à la classe **Moteur** une variable membre appelée **fichierLog** et de type **ofstream**. Cette variable membre n'aura aucun accesseur car seulement utilisée par les méthodes de la classe **Moteur**. L'objet **fichierLog** sera instancié dans le constructeur de la classe **Moteur** et on l'associera au fichier texte « **traces.log** » qui sera ouvert en mode « écriture à la fin ».

Ensuite, dans chaque méthode de traitement de la classe **Moteur**, vous devrez utiliser l'objet **fichierLog** pour écrire dans le fichier de traces. Celui-ci pourra ainsi être consulté par n'importe quel éditeur de texte.

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, la classes **Moteur** et **UI** pourraient être mises à jour afin d'en tenir compte.

Bon travail 😊 !