

Laboratoire de Programmation en C++

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

Année académique 2017-2018

Inpres Paint

**Anne Léonard
Denys Mercenier
Patrick Quettier
Claude Vilvens
Jean-Marc Wagner**

0. Introduction

0.1 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++ ».

Quelque 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 2017 (sur base d'une liste de questions fournies en novembre et à préparer) et coté sur 20;
- ♦ laboratoire (cet énoncé) : 2 évaluations (aux dates précisées dans l'énoncé de laboratoire), chacune cotée sur 20; la moyenne arithmétique pondérée (30% pour la première partie et 70% pour la seconde partie) de ces 2 cotes 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) Chacun des membres d'une équipe d'étudiants doit être capable d'expliquer et de justifier l'intégralité du travail (pas seulement les parties du travail sur lesquelles il aurait plus particulièrement travaillé)

2) En 2^{ème} session, un **report de note** est possible pour chacune des deux notes de laboratoire ainsi que pour la note de théorie **pour des notes supérieures ou égales à 10/20**.

Toutes les évaluations (théorie ou laboratoire) ayant des **notes inférieures à 10/20** sont à **représenter dans leur intégralité**.

3) Les consignes de présentation des dossiers de laboratoire sont fournies par les différents professeurs de laboratoire via leur centre de ressources

0.2 Le contexte : Inpres Paint

Les travaux de Programmation Orientée Objets (POO) C++ consistent à réaliser une ébauche de logiciel de dessin vectoriel. Contrairement à un logiciel comme Microsoft Paint qui considère une image comme un ensemble de pixels de différentes couleurs, un logiciel de dessin vectoriel considère une image comme un ensemble de formes (points, droites, rectangles, ...) ayant chacune une certaine couleur et une profondeur (certaines formes se trouvent derrière ou devant d'autres).

Il s'agit donc ici de mettre en place les bases fonctionnelles d'un tel logiciel. A terme, il sera néanmoins possible d'ouvrir une fenêtre graphique permettant d'encoder et de dessiner les différentes formes constituant l'image.

0.3 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 1^{er} quart puis de conforter vos acquis au 2^{ème} quart. 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;

Le dossier est prévu à priori pour une équipe de deux étudiants qui devront donc se coordonner intelligemment et se faire confiance. Il est aussi possible de présenter le travail seul (les avantages et inconvénients d'un travail par deux s'échangent).

Il s'agit bien d'un laboratoire de C++ sous UNIX. La machine de développement sera Sunray. Même s'il n'est pas interdit (que du contraire) de travailler sur un environnement de votre choix (**Dev-C++** sur PC/Windows sera privilégié car compatible avec C++/Sunray – à la rigueur Visual C++ sous Windows, g++ sous Linux, etc ...) à domicile, seul le code compilable sous Sunray sera pris en compte !!! Une machine virtuelle possédant exactement la même configuration que celle de Sunray sera mise à la disposition des étudiants lors des premières séances de laboratoire.

Un petit conseil : *lisez bien l'ensemble de l'énoncé* avant de concevoir (d'abord) ou de programmer (après) une seule ligne ;-). Dans la deuxième partie (au plus tard), prévoyez une schématisation des diverses classes (diagrammes de classes **UML**) et élaborer d'abord "sur papier" (donc sans programmer directement) les divers scénarios correspondant aux fonctionnalités demandées.

0.4 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 début novembre**), vous vous situez en tant que « programmeur créateur de classes ». On va donc vous fournir une série de 7 jeux de test (les fichiers Test1.cpp, Test2.cpp, ..., Test7.cpp) qui contiennent une fonction main() et qui vous imposeront le comportement (l'interface) de vos classes.

Dans la deuxième partie du laboratoire (**de début 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.

0.5 Planning et contenu des évaluations

a) Evaluation 1 (continue) :

Porte sur : les 5 premiers jeux de tests (voir tableau donné plus loin).

Date de remise du dossier : le 1^{er} lundi du 2^{ème} quart à 12h00 au plus tard.

Modalités d'évaluation : à partir du 1^{er} lundi du 2^{ème} quart selon les modalités indiquées par le professeur de laboratoire.

b) Evaluation 2 (examen de janvier 2017) :

Porte sur : les classes développées dans les jeux de tests 6 et 7, et le développement de l'application finale (voir tableau donné plus loin).

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

Modalités d'évaluation : selon les modalités fixées par le professeur de laboratoire.

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**.

1. Première Partie : Création de diverses briques de base nécessaires (Jeux de tests)

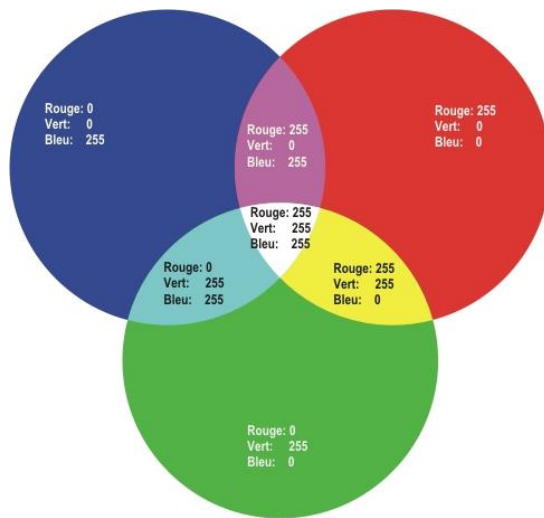
Points principaux de l'évaluation	Subdivisions
EVALUATION 1 (Jeux de tests 1 à 5) → 1^{er} lundi du 4^{ème} quart	
<ul style="list-style-type: none"> Jeu de tests 1 : Implémentation d'une classe de base (Couleur) 	Constructeurs, destructeurs
	Getters, Setters et méthode Affiche
	Fichiers Couleur.cpp, Couleur.h et makefile
<ul style="list-style-type: none"> Jeu de tests 2 : Agrégation entre classes (classe Forme) 	Agrégation par valeur (classe Point)
	Agrégation par référence (classe Couleur)
	Variables membres statiques (type int et Couleur)
<ul style="list-style-type: none"> Jeu de tests 3 : Surcharge des opérateurs de la classe Couleur 	Opérateur =
	Opérateurs +
	Opérateur -
	Opérateurs ==, < et >
	Opérateurs << et >>
	Opérateurs ++ et --
<ul style="list-style-type: none"> Jeu de tests 4 : Héritage et virtualité 	Classe abstraite Forme
	Classes dérivées Pixel , Ligne et Rectangle
	Test de la virtualité et du down-casting
<ul style="list-style-type: none"> Jeu de tests 5 : Exceptions 	BaseException
	InvalidColorException qui hérite de BaseException
	Utilisation correcte de try , catch et throw
EVALUATION 2 (Jeux de tests 6 à 7) → Examen de Janvier 2018	
<ul style="list-style-type: none"> Jeu de test 6 : Containers génériques 	Classe abstraite ListeBase
	Classe Liste (→ int et Couleur)
	Classe ListeTriee (→ int et Couleur)
	Itérateur : classe Iterateur
<ul style="list-style-type: none"> Jeu de test 7 : Flux 	Méthodes Save() et Load() de Point
	Méthodes Save() et Load() de Couleur
	Méthodes Save() et Load() de Ligne

1.1 Jeu de tests 1 (Test1.cpp) :























Une première classe

a) Description des fonctionnalités de la classe

Un des éléments principaux de l'application est évidemment la notion de couleur. Sans entrer dans des notions théoriques avancées, une couleur est représentée sur ordinateur par un mélange de trois couleurs de base : le rouge, le vert et le bleu. Une couleur est donc caractérisée par 3 entiers compris entre 0 et 255 représentant les quantités respectives de rouge, vert et bleu constituant la couleur :



Exemples de couleurs définies par leur code RVB

Nom de la couleur		Code RVB			Nom de la couleur		Code RVB		
Rouge		255	0	0	Jaune		255	255	0
Vert		0	255	0	Lavande		150	131	236
Bleu		0	0	255	Magenta		255	0	255
Blanc		255	255	255	Marine		3	34	76
Noir (absence de couleur)		0	0	0	Marron		88	41	0
Argent (gris léger)		206	206	206	Olive		112	141	35
Bleu de cobalt		34	66	124	Pêche		253	191	183
Bordeaux		109	7	26	Rose		253	108	158
Carotte		244	102	27	Saumon		248	152	85
Cyan		0	255	255	Vert kaki		121	137	51
Grenadine		233	56	63	Violet		127	0	255

Notre première classe, la classe **Couleur**, sera donc caractérisée par :

- **rouge, vert, bleu** : trois entiers (**int**) compris entre 0 et 255 caractérisant la couleur.
- Un **nom** : une chaîne de caractères allouée dynamiquement (**char ***) en fonction du texte qui lui est associé. Il s'agit du nom donné à la couleur.

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

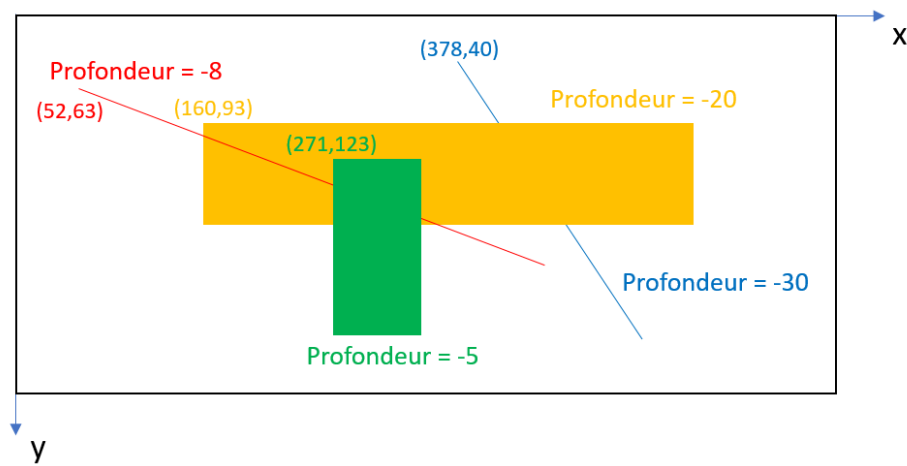
Veillez à tracer (cout << ...) vos constructeurs et destructeurs pour que vous puissiez vous rendre compte de quelle méthode est appelée et quand elle est appelée.

On vous demande de créer pour la classe Couleur (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.

1.2 Jeu de tests 2 (Test2.cpp) :

Associations entre classes : agrégations + Variables statiques

Il s'agit à présent de représenter en mémoire les objets graphiques ou plutôt les formes géométriques (ligne, rectangle, pixel, ...) que notre application sera capable de gérer. Tout d'abord, une forme géométrique est positionnée dans l'image, cette position sera caractérisée par ses coordonnées (x,y). Une forme possède aussi une couleur mais également ce que l'on appelle une *profondeur*. Cette profondeur (valeur entière généralement négative) nous indique quelles formes doivent être dessinées «derrière» et quelles formes doivent être dessinées «par-dessus». Exemple :



Dans cet exemple, la ligne bleue a une position (1^{ère} extrémité) située en (x,y) = (378,40) et une profondeur de -30, elle est donc dessinée «derrière» toutes les autres formes qui ont une profondeur plus grande. Le rectangle vert plein a une position (coin en haut à gauche) égale à (x,y) = (271,123) et une profondeur de -5, il est dessiné par-dessus toutes les autres formes car il a la plus grande profondeur.

a) La classe Point (Essai1())

Commençons par modéliser la notion de position d'une forme. On vous demande de créer la classe **Point** contenant

- Deux variables **X** et **Y** de type **int** représentant les coordonnées d'un point dans une image.
- Un constructeur par défaut, un de copie et un d'initialisation (voir jeu de tests), ainsi qu'un destructeur (dans la suite, nous n'en parlerons plus → toute classe digne de ce nom doit au moins contenir un constructeur par défaut et un de copie, ainsi qu'un destructeur).
- Les méthodes getXXX()/setXXX() associées,
- Une méthode Affiche() permettant d'afficher les coordonnées du Point.

b) La classe Forme (Essai2())

Il s'agit à présent de représenter en mémoire une forme géométrique dont nous avons parlé plus haut. On vous demande de créer la classe **Forme** contenant

- Un **id** : une chaîne de caractères allouée dynamiquement (**char ***) en fonction du texte qui lui est associé. Il s'agit de l'identifiant de la forme.
- Une variable **position** de type **Point**.
- Un pointeur couleur de type **Couleur*** pointant vers un objet de type Couleur si une couleur a déjà été attribuée à cette forme, ou valant NULL sinon.
- Une variable **profondeur** de type **int**.
- Deux constructeurs d'initialisation (voir jeu de tests). Une classe peut avoir plusieurs constructeurs d'initialisation. Par défaut ou par initialisation partielle, un objet forme ne possède pas de couleur (NULL).
- Les méthodes `getXXX()/setXXX()` associées aux variables membres.
- La méthode `Affiche()` qui affiche toutes les caractéristiques de la forme géométrique.

Bien sûr, les classes **Point** et **Forme** doivent posséder leurs propres fichiers .cpp et .h.

La classe **Forme** contenant une variable membre dont le type est une autre classe, on parle d'**agrégation par valeur**, l'objet de type **Point** fait partie intégrante de l'objet **Forme**.

La classe **Forme** possède également un pointeur vers un objet de la classe **Couleur**. Elle ne contient donc pas l'objet **Couleur** en son sein mais seulement un pointeur vers un tel objet. On parle d'**agrégation par référence**.

c) Variables statiques compteur d'objets instanciés (Essai3())

Afin de réaliser un « monitoring » de la mémoire, il serait intéressant de connaître en permanence le nombre d'objets de chaque classe instanciés à un moment donné. Pour cela, on demande d'ajouter à chacune des classes **Couleur**, **Point** et **Forme**,

- Une variable **compteur**, **statique**, **privée**, de type **int**. Celle-ci contient en permanence le nombre d'objets instanciés pour la classe en question. Elle est donc incrémentée automatiquement à chaque création d'objet et décrémentée à chaque destruction.
- Une méthode `getCompteur()`, **statique**, **publique**, de type **int**. Cette méthode retourne la valeur du compteur.

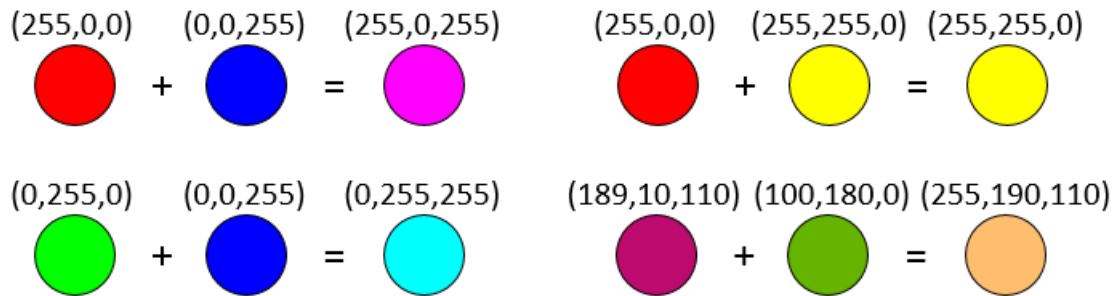
d) Variables statiques de type Couleur (Essai4())

On se rend bien compte qu'il y a des couleurs qui apparaissent plus souvent que d'autres, comme le rouge, le vert ou le bleu. 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 couleurs particulières. Dès lors, on vous demande d'ajouter, à la classe **Couleur**, **3 variables membres publiques**, appelées **ROUGE**, **VERT** et **BLEU**, **statiques**, **constantes** de type **Couleur** et ayant les caractéristiques respectives (`r=255,v=0,b=0,nom="Rouge"`), (`r=0,v=255,b=0,nom="Vert"`) et (`r=0,v=0,b=255,nom="Bleu"`). Voir jeu de tests.

1.3 Jeu de tests 3 (Test3.cpp) :

Extension des classes existantes : 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. En particulier, nous allons tout d'abord mettre en place des opérateurs permettant de réaliser des mélanges de couleurs (on parle de « synthèse additive de couleurs »), permettant de créer de nouvelles couleurs. Ce mélange consiste simplement à additionner les composantes (rouge, vert, bleu) de chaque couleur. Exemples :



Remarquez que ce type de mélange n'est pas intuitif. On pourrait croire, par exemple, que le mélange de rouge et de jaune donne de l'orange, mais il n'en est rien. De plus, les composantes (rouge, vert, bleu) obtenues après addition ne peuvent jamais dépasser la valeur maximale de 255. Néanmoins, dans tous les cas, on peut observer que la nouvelle couleur est toujours plus (ou égal) « claire » que les deux couleurs que l'on additionne. Cette notion de « clarté » porte le nom de **luminance** de la couleur. La luminance d'une couleur est simplement définie par la moyenne arithmétique de ses trois composantes. Par exemple,

Luminance du bleu = $(0 + 0 + 255) / 3 = 85$

Luminance du cyan = $(0 + 255 + 255) / 3 = 170$

Plus la luminance est grande, plus la couleur est claire et plus la luminance est petite, plus la couleur est foncée. La luminance est toujours comprise entre 0 (le noir) et 255 (le blanc).

Dès lors, avant tout chose, on vous demande d'ajouter à la classe **Couleur** :

- La méthode **getLuminance()**, de type **int**, qui retourne la luminance de la couleur (arrondie à l'unité la plus proche)

a) Surcharge de l'opérateur = de la classe Couleur : Essai1()

Dans un premier temps, on vous demande de surcharger l'opérateur = de la classe Couleur, permettant d'exécuter un code du genre :

```
Couleur c, c1(...);  
c = c1 ;
```

b) Surcharge de l'opérateur (Couleur + Couleur) de la classe Couleur : Essai2()

Il s'agit à présent de mettre en place l'opérateur permettant de réaliser le mélange de couleurs dont nous avons parlé plus haut. On vous demande donc de surcharger l'opérateur + de la classe Couleur permettant d'exécuter un code du genre :

```
Couleur c1(244,102,27,"Carotte"), c2(112,141,35,"Olive"), c3 ;  
c3 = c1 + c2 ; // c3 correspond alors à (255,243,62,"Melange de Carotte et de Olive")
```

Lorsque les deux couleurs possèdent un nom, le nom de la nouvelle couleur sera « Mélange de XXX et de YYY » où XXX et YYY sont les noms des deux couleurs que l'on additionne. Si une des couleurs au moins n'a pas de nom, la nouvelle couleur aura le nom « Melange inconnu ».

c) Surcharge des opérateurs (Couleur + int) de la classe Couleur : Essai3()

Il s'agit ici de mettre en place le(s) opérateur(s) permettant d'**augmenter la luminance** d'une couleur (de l'éclaircir donc). Pour cela, il suffit d'ajouter **une même valeur entière** aux 3 composantes de la couleur de départ, de nouveau sans que les composantes ne dépassent 255. Exemple : (230,100,127) + 30 = (255,130,157). On vous demande donc de programmer le(s) opérateur(s) + de la classe Couleur permettant d'exécuter un code du genre :

```
Couleur c1,c2(110,200,45);  
c1 = c2 + 20 ;  
c1 = c1 + 10 ;  
c2 = 10 + c1 ;  
c2 = 30 + c2 ;
```

Si la couleur de départ possède le nom « XXX », la nouvelle couleur aura le nom « XXX clair(20) » si on a ajouté 20 à la couleur de départ. Si la couleur de départ n'a pas de nom, la nouvelle couleur n'aura pas de nom non plus.

d) Surcharge de l'opérateur (Couleur - int) de la classe Couleur : Essai4()

Il s'agit ici de mettre en place l'opérateur permettant de **diminuer la luminance** d'une couleur (de l'assombrir donc). Pour cela, il suffit de soustraire **une même valeur entière** aux 3 composantes de la couleur de départ, mais sans que les composantes ne deviennent négatives. Exemple : (130,50,18) - 30 = (100,20,0). On vous demande donc de programmer l'opérateur - de la classe Couleur permettant d'exécuter un code du genre :

```
Couleur c1,c2(110,200,45);  
c1 = c2 - 20 ;  
c1 = c1 - 10 ;
```

Si la couleur de départ possède le nom « XXX », la nouvelle couleur aura le nom « XXX fonce(20) » si on a soustrait 20 à la couleur de départ. Si la couleur de départ n'a pas de nom, la nouvelle couleur n'aura pas de nom non plus.

e) Surcharge des opérateurs de comparaison de la classe Couleur : Essai5()

Il s'agit ici de mettre en place les opérateurs permettant de comparer deux couleurs sur leur valeur de luminance. Dans ce but, on vous demande de surcharger les opérateurs <, > et == de la classe Couleur permettant d'exécuter un code du genre

```
Couleur c1,c2;
...
if (c1 > c2) cout << "c1 est plus claire que c2" ;
if (c1 == c2) cout << autre message;
if (c1 < c2) ...
```

f) Surcharge des opérateurs d'insertion << et d'extraction >> : Essai6()

On vous demande à présent de surcharger les opérateurs << et >> de la classe Couleur, ce qui permettra d'exécuter un code du genre :

```
Couleur c1;
cout << "Entrez une couleur :";
cin >> c1; // permet d'encoder une couleur sous la forme d'une chaîne
           // de caractères "248 152 85 Saumon"
cout << c1 ; // affiche la chaîne de caractères "[248,152,85 (Saumon)]"
```

g) Surcharge des opérateurs ++ et -- de classe Couleur : Essai7() et Essai8()

On vous demande de programmer les opérateurs de post et pré-in(dé)crémentation de la classe Couleur. Ceux-ci in(dé)crémenteront un objet Couleur **d'une luminance de 10** (10 ajouté ou soustrait à chaque composante de la couleur tout en restant compris entre 0 et 255). Cela permettra d'exécuter le code suivant :

```
Couleur c(150,131,236,"Lavande") ;

cout << ++c << endl ; // c vaut à présent (160,141,246)
cout << c++ << endl ; // c vaut à présent (170,151,255)
cout << --c << endl ; // c vaut à présent (160,141,245)
cout << c-- << endl ; // c vaut à présent (150,131,235)
```

1.4 Jeu de tests 4 (Test4.cpp) :

Associations de classes : héritage et virtualité

On se rend vite compte que la notion de forme géométrique est un peu limitée. En effet, une position seule ne suffit pas décrire une forme géométrique. Nous allons donc *spécialiser* notre classe **Forme** afin d'obtenir des objets géométriques plus spécifiques comme les pixels, les lignes ou les rectangles. De plus, instancier la classe **Forme** n'a pas de sens car elle manque d'information. Par contre, un pixel, une ligne ou un rectangle possède toutes les caractéristiques de base d'une forme telle que nous l'avons imaginée. Toutes ces considérations nous mènent à concevoir la petite hiérarchie de classes décrite ci-dessous.

a) La classe **Forme** devient abstraite

L'idée est de concevoir une hiérarchie de classes, par héritage, dont la classe de base regroupera les caractéristiques communes de toutes les formes géométriques que nous considérerons. Cette classe de base sera évidemment notre classe **Forme** mais modifiée de la manière suivante :

- On lui ajoute la méthode virtuelle pure **getInfos()**, de type **char***, qui retournera, dans les classes héritées, une chaîne de caractères fournissant toutes les informations de la forme (voir plus bas). La classe **Forme** devient donc abstraite.
- On lui ajoute une variable membre **infos**, de type **char***, qui pointera vers la chaîne de caractères, allouée dynamiquement en fonction de ce qu'elle contient, calculée et retournée à chaque appel de la méthode **getInfos()**.

b) La classe **Pixel** : Essai1()

On vous demande de programmer la classe **Pixel**, héritant de la classe **Forme** et qui :

- Possède ses propres constructeurs et opérateur <<
- Redéfinit la méthode **getInfos()** de telle sorte qu'elle retourne une chaîne de caractères ayant le format suivant :

```
[PIXEL : Position(10,40),Couleur(255,0,0,Rouge),Profondeur=-10]
```

pour un pixel située aux coordonnées (10,40), de couleur rouge et de profondeur -10

c) La classe **Ligne** : Essai2()

On vous demande de programmer la classe **Ligne**, qui hérite de la classe **Forme**, et qui présente en plus :

- Une variable **extremite**, de type **Point**, qui représente les coordonnées de l'extrémité de la ligne, son origine étant sa position.
- Ses propres constructeurs et opérateur <<
- la méthode **getInfos()** qui retourne une chaîne de caractères ayant le format suivant :

```
[LIGNE : Position(30,90),Extremite(100,120),Couleur(0,255,0),Profondeur=-10]
```

pour une ligne d'origine (30,90), d'extrémité (100,120), de couleur (0,255,0) et de profondeur 10.

d) La classe Rectangle : Essai3()

On vous demande de programmer la classe **Rectangle**, qui hérite de la classe **Forme**, et qui présente en plus :

- Deux variables **dimX** et **dimY**, de type **int**, qui représentent les dimensions horizontale (la « largeur ») et verticale (la « hauteur ») du rectangle. La position correspond au coin supérieur gauche du rectangle.
- Une variable **rempli**, de type **bool**, indiquant si le rectangle devra être rempli ou non lors du dessin. Dans la négative, on ne dessinera que les bords du rectangle (voir 2^{ème} partie du dossier).
- Ses propres constructeurs et opérateur <<
- La méthode **getInfos()** qui retourne une chaîne de caractères ayant le format suivant :

```
[RECTANGLE : Position(30,90),DimX=200,DimY=140,Rempli=1,Couleur(0,255,0),Profondeur=-10]
```

pour un rectangle dont le coin supérieur gauche se situe en (30,90), de largeur 200, de hauteur 140, rempli, de couleur (0,255,0) et de profondeur -10.

e) Mise en évidence de la virtualité et du down-casting : Essai4() et Essai5()

La méthode **getInfos()** étant virtuelle, on vous demande de :

- comprendre et savoir expliquer le code de l'essai 4 mettant en évidence la virtualité de la méthode **getInfos()**.
- comprendre et savoir expliquer le code de l'essai 5 mettant en évidence le down-casting et le dynamic-cast du C++.

1.5 Jeu de tests 5 (Test5.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 lié à l'erreur. Elle sera lancée lorsque l'id (identifiant) d'une Forme est invalide. Pour que l'identifiant d'un **Pixel** soit valide, il doit

- Commencer par la lettre 'P'
- Contenir au moins 2 caractères
- Ne contenir que des chiffres (mis à part la première lettre 'P').

Par exemple, si pix est un objet de la classe Pixel, pix.setId("P") ou p.setId("PX34") lancera une exception avec un message du genre « Identifiant invalide ! ». Un exemple d'id valide est "P17". En ce qui concerne les identifiants de **Ligne** ou de **Rectangle**, il en est de même sauf que la première lettre doit être un 'L' ou un 'R' selon le cas. Vous devez donc **surcharger la méthode setId()** dans les classes Pixel, Ligne et Rectangle tout en faisant attention à la **virtualité de la méthode... A vous de voir !!! Virtuelle ou pas ?**

- **InvalidColorException** : lancée lorsque l'on tente de créer ou modifier une couleur avec une composante (au moins) qui n'est pas comprise entre 0 et 255. **Cette classe va hériter directement de BaseException** mais possèdera en plus :
 - une méthode **rougeValide()**, de type **bool**, retournant true si la composante rouge est valide lors la création/modification de la couleur, false sinon.
 - une méthode **vertValide()**, de type **bool**, retournant true si la composante verte est valide lors la création/modification de la couleur, false sinon.
 - une méthode **bleuValide()**, de type **bool**, retournant true si la composante bleue est valide lors la création/modification de la couleur, false sinon.

Par exemple, si c1 est un objet de la classe Couleur, c1.setRouge(300) lancera une InvalidColorException e et e.rougeValide() retournera false, e.vertValide() retournera true et e.bleuValide() retournera true, le message pouvant être « Modification Couleur Impossible ! ». Si maintenant, on tente de créer une couleur c2 en utilisant Couleur c2(300,300,100), ce constructeur lancera une InvalidColorException e et e.rougeValide() retournera false, e.vertValide() retournera false et e.bleuValide() retournera true, le message pouvant être « Création Couleur Impossible ! ». Il sera ainsi possible au programmeur de mieux préciser la cause de l'erreur et d'agir en conséquence.

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 Test5.cpp** → utilisation de **try**, **catch** et **throw**), mais également dans l'application finale.

1.6 Jeu de tests 6 (Test6.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 toutes les formes géométriques d'une image ou toutes les couleurs utilisées. 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** **T* 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, ...) et de retourner l'adresse de l'objet T inséré dans la liste. **Attention !!!** Le fait que la méthode insere retourne un pointeur (non constant) permettra de modifier l'objet T pointé. Une modification d'une des variables membres de cet objet sur laquelle portent les opérateurs de comparaison <, >, ou == pourrait endommager une liste triée (qui ne le serait donc plus ☹). A utiliser avec prudence !
- 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

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

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 **Couleur**. Ceux-ci devront bien sûr être triés par ordre croissant de luminance.

e) **Parcourir et modifier une liste : l'itérateur de liste**

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 d'en modifier un et encore moins d'en supprimer un. 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 ListeTriee), 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 &** qui retourne l'adresse de l'élément (objet T) pointé par l'itérateur.
- **remove()** qui retire de la liste et retourne l'élément pointé par l'itérateur.

L'application finale fera un usage abondant de la classe ListeTriee. 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.

1.7 Jeu de tests 7 (Test7.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.

La classe Ligne se sérialise elle-même : Essai1(), Essai2() et Essai3()

On demande de compléter la classe **Ligne** avec les deux méthodes suivantes :

- ♦ **Save(ofstream & fichier) const** permettant d'enregistrer sur flux fichier toutes les données d'une ligne (id, position, couleur (**!!! Temporairement car dans l'application finale, ce ne sera pas l'objet Couleur qui sera enregistré sur disque mais bien une « référence » (son nom) d'un objet Couleur qui sera enregistré indépendamment !!!**), profondeur et extrémité) 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 ligne 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 **Point**, **Couleur**, et **Forme** 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 de la classe Ligne 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.

2. Deuxième Partie : Développement de l'application

Points principaux de l'évaluation	subdivisions
EVALUATION 2 (Développement de l'application) → Examen de Janvier 2017	
• <u>Classe SmartPointer :</u>	Constructeurs
	Destructeur et méthode Delete()
	Surcharge des opérateurs
	ListeTrie<SmartPointer<Forme> >
	Test 8 OK ?
• <u>Classe Dessin :</u>	Liste<Couleur> et ListeTrie<SmartPointer<Forme> >
	Méthodes ajouteCouleur() et ajouteForme()
	Sérialisation : méthodes Save() et Load()
	Lecture fichiers csv : méthodes importXXX()
	Méthode virtuelle pure dessine() de Forme et classes héritées
	Méthode dessine() de Dessin : parcours des formes (itérateur)
	Gestion correcte des exceptions ?
• <u>Fonctionnalités du Menu</u>	Dessin : nouveau, ouvrir, enregistrer, fermer
	Gestion générale
	Palette de couleurs : ajouter/modifier/supprimer/afficher/csv
	Formes : ajouter/supprimer/modifier couleur/afficher/csv
• <u>Bonus</u>	Ajout d'autres formes ?
	Gestion du clic de la souris ?

3.1 L'application proprement dite : **Premier aperçu**

Au démarrage, l'application propose le menu unique suivant :

```
*****
***** Inpres Paint *****
*****
0. Quitter
1. Nouveau Dessin
2. Ouvrir Dessin
3. Enregistrer Dessin
4. Fermer Dessin
Gestion générale *****
5. Afficher les paramètres du dessin (taille et couleur de fond)
6. Spécifier une couleur de fond
7. Afficher l'état de la mémoire
Gestion de la palette de couleurs *****
8. Ajouter une couleur
9. Modifier une couleur
10. Supprimer une couleur
11. Afficher toutes les couleurs de la palette
12. Importer une palette de couleurs à partir d'un fichier csv
Gestion des formes géométriques *****
13. Ajouter une forme
14. Supprimer une forme
15. Modifier la couleur d'une forme
16. Afficher les infos de toutes les formes
17. Importer un ensemble de formes à partir d'un fichier csv
```

Comme nous allons le voir, l'application sera capable de gérer deux types de fichiers :

- Des **fichiers binaires** : un dessin sera enregistré dans un fichier binaire d'extension « .des ». Par exemple, « MonBeauDessin.des ».
- Des **fichiers textes/csv** : il s'agit de fichiers textes qui permettront à l'utilisateur d'importer un ensemble de couleurs et/ou un ensemble de formes dans son dessin en cours.

Au démarrage, l'application ne contient aucun dessin. Il faudra soit créer un nouveau dessin, soit ouvrir un dessin existant avant de pouvoir travailler. Quand un dessin est « ouvert », une fenêtre graphique apparaît (voir plus loin) et le dessin en cours y est dessiné. Dès que l'on ferme un dessin, cette fenêtre se ferme également.

Chaque fonctionnalité du menu sera décrite en détails plus loin. Avant cela, nous allons décrire les classes supplémentaires à développer afin de répondre à nos besoins.

3.2 Des pointeurs « intelligents » : les « SmartPointer »

Il est clair que notre application va comporter des conteneurs avec toutes les formes composant un dessin. On pourrait imaginer un conteneur par type de forme (Pixel, Ligne, Rectangle) mais cela ne serait pas efficace. En effet, pour un affichage correct, toutes les formes doivent être triées par ordre croissant de leur profondeur. Avec un conteneur de chaque type, les rectangles seraient triés entre eux, ainsi que les lignes et les pixels, mais les formes ne seraient pas triées globalement !

L'idée est alors de créer une seule liste triée template de « Forme ». Mais cela est impossible car la classe Forme est abstraite... Une solution hasardeuse serait alors de créer une liste triée de pointeurs de type Forme (ListeTriee<Forme*>), pointeurs qui pourraient pointer vers des objets de type Rectangle, Ligne ou Pixel. Mais... la liste serait alors triée sur la valeur de ces pointeurs et non sur la profondeur !!!

La solution à ce dernier problème consiste donc à encapsuler ce pointeur dans une classe généralement connue sous le nom de « SmartPointer » et dont les opérateurs de comparaison sont surchargés de telle sorte que l'on compare les objets pointés et non les valeurs de pointeur.

On vous demande donc de développer la classe **template SmartPointer** dont la structure est la suivante (afin de mettre au point cette classe, le fichier **Test8.cpp** vous est fourni) :

```
template<class T> class SmartPointer
{
    private:
        T * val;
    ...
};
```

Et qui contient les méthodes suivantes

- **Constructeurs** par défaut, d'initialisation et de copie,
- Un **destructeur**. Mais attention ! Ce destructeur ne doit pas libérer la zone mémoire de l'objet T pointé par val, car celle-ci est peut-être statique.
- Une méthode **Delete()** qui libère la mémoire associée à l'objet T pointé
- Deux **opérateurs** =, un qui prend en paramètre un SmartPointer, l'autre prenant en paramètre l'adresse d'un objet T
- Une méthode **T* getVal()** qui retourne la valeur du pointeur
- La surcharge de l'**opérateur *** retournant, par référence, l'objet T pointé par val
- La surcharge de l'**opérateur ->** qui retourne, comme getVal(), la valeur du pointeur
- Les **opérateurs** de comparaison <, >, ==, ... permettant de comparer les objets pointés (et non les pointeurs) en faisant appel aux opérateurs de comparaison de la classe T
- L'**opérateur <<** affichant la valeur du pointeur

Il est à présent possible de créer notre conteneur de formes : **ListeTrie<SmartPointer<Forme> >** qui contient les pointeurs vers toutes les formes qui seront ajoutées (voir Test8.cpp).

Remarque importante : Faites attention au fait que le destructeur de ListeTrie ne désallouera pas les formes !!! Il sera nécessaire, de balayer la liste avec l'itérateur afin d'appeler la méthode **Delete()** sur chaque élément de la liste (voir Test8.cpp de nouveau).

3.3 Un conteneur pour nos conteneurs : La classe Dessin

Un dessin est évidemment composé de *formes* géométriques et de couleurs (l'ensemble des couleurs présentes dans un dessin est appelé *palette de couleurs*) mais ce n'est pas suffisant pour le définir complètement. Il faut également préciser la *taille* de notre dessin (largeur et hauteur de l'image en pixels), ainsi que la *couleur de fond* sur lequel les formes seront dessinées.

On vous demande donc de regrouper toutes ces données au sein de la même classe :

```
class Dessin
{
    private :
        int largeur;
        int hauteur;
        Couleur fond;
        Liste<Couleur> palette;
        ListeTrie<SmartPointer<Forme> > formes;

    public :
        Dessin() ;
        Dessin(int l,int h,const Couleur& f) ;
        void ajouteCouleur(const Couleur& c) ;
        void ajouteForme(Forme *pf,const char* nomCouleur) ;
        ...
} ;
```

Cette classe comporte notamment :

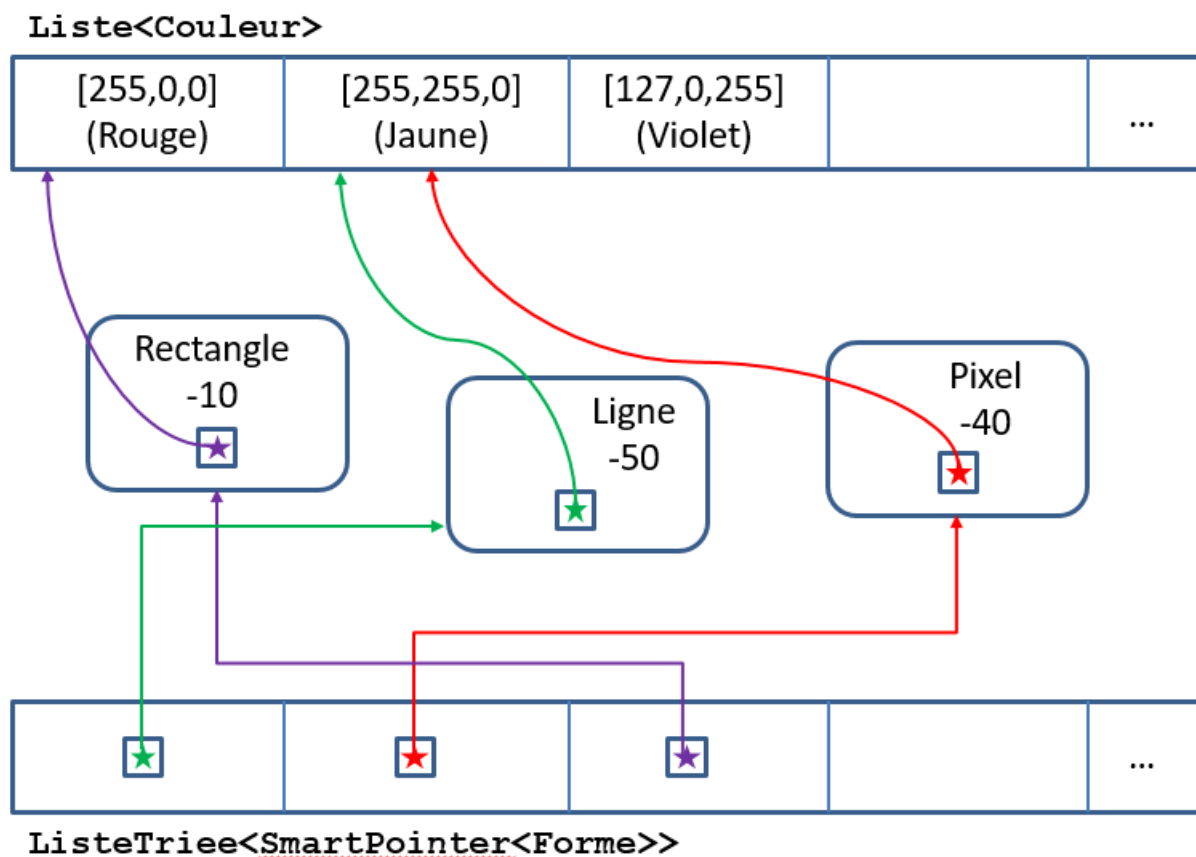
- L'ensemble des données (variables membres) relatives au dessin.
- Des **constructeurs**. Notez qu'une fois l'objet construit, il ne sera plus possible de modifier les dimensions de l'image.
- Une méthode **ajouteCouleur** qui permettra d'ajouter une couleur à la **palette** de couleurs. Attention, il ne sera pas possible d'ajouter deux couleurs de même nom. Le nom identifie donc une couleur de manière unique. Dans le cas où l'on essaierait tout de même de le faire, la méthode devra lancer une exception de votre choix.
- Une méthode **ajouteForme** qui permettra d'ajouter, au conteneur **formes**, une forme allouée dynamiquement avant l'appel de la méthode. Cet objet n'aura au départ aucune couleur. Le second paramètre de la méthode ajouteForme permettra de lui attribuer une couleur faisant partie de la palette (la couleur devra donc déjà exister !). L'objet Forme pointera alors vers

l'objet Couleur trouvé dans la palette sur base de son nom (voir schéma ci-dessous). Si aucune couleur correspondant à `nomCouleur` n'est présente dans la palette de couleurs, la méthode devra lancer une **exception** de votre choix. De même si on tente d'ajouter une forme dont l'identifiant existe déjà, une **exception** sera lancée également.

- Un **destructeur** qui balayera la liste de formes afin de les désallouer correctement (appel à `Delete()` de la classe `SmartPointer`).
- ...D'autres méthodes que vous devrez développer afin de répondre aux fonctionnalités demandées (voir plus loin).

Schéma des conteneurs de la classe Dessin et leurs relations

Le schéma suivant présente les différents conteneurs de données, ainsi que les pointeurs allant des formes vers les couleurs :



Sur ce schéma, nous voyons

- Une palette comportant 3 couleurs dont 2 sont utilisées actuellement
- Un rectangle de couleur « Rouge »
- Une ligne et un pixel de même couleur « Jaune »
- La liste triée de formes est bien triée selon les valeurs de profondeur : -50, -40, -10

3.4 Enregistrement sur disque : La classe Dessin se sérialise elle-même

Comme nous l'avons dit précédemment, un dessin devra être sérialisé dans un **fichier binaire** (utilisation des méthodes write et read) d'extension « .des ». Un tel fichier sera structuré de la manière suivante :

1. Taille de l'image : **largeur** et **hauteur**
2. Couleur de fond : l'objet **fond**
3. La palette de couleur : pour cela, on écrira tout d'abord le **nombre de couleurs**, puis ensuite **chacun des objets Couleur** présents dans le conteneur palette. Lors de la lecture, on saura alors exactement combien d'objets Couleur lire.
4. Les formes géométriques : pour cela, on écrira tout d'abord le **nombre de formes**, puis ensuite **chacun des objets « Forme »**. Le **caractère 'R'** devra précéder un objet Rectangle, le **caractère 'L'** devra précéder un objet Ligne et le **caractère 'P'** devra précéder un objet Pixel. Lors de la lecture, on saura alors exactement combien de formes lire et le caractère lu avant chaque forme permettra de savoir quel type de forme allouer.

On vous demande donc d'ajouter à la classe Dessin :

- La méthode **Save(const char* nomFichier)** qui enregistrera toutes les données du dessin selon la structure décrite ci-dessus.
- La méthode **Load(const char* nomFichier)** qui lira sur disque toutes les données d'un dessin.

Ces deux méthodes devront appeler les méthodes Save et Load de tous les objets intervenants (Couleur, Point, Forme, Ligne, ...). Chaque objet va donc se sérialiser lui-même.

Plusieurs remarques importantes :

- Lors de l'**écriture d'une forme**, on ne devra pas écrire l'objet Couleur sur disque (car celui-ci aura déjà été écrit au préalable). Il faudra donc juste écrire le nom de la couleur.
- Lors de l'**écriture des formes**, le conteneur formes est parcouru à l'aide de l'itérateur. Mais comment savoir quel type d'objet est pointé par le pointeur Forme* afin de pouvoir écrire le bon caractère 'R', 'L' ou 'P' ? On vous demande donc d'utiliser le dynamic-cast du C++. De plus il serait judicieux de rendre virtuelle la méthode Save() de Forme... Pourquoi ☺ ?
- Lors de la **lecture d'une forme**, seul le nom de la couleur est récupéré. Mais il faut encore remettre à jour le pointeur couleur ! Pour cela, il suffit de passer la palette de couleurs en paramètre à la méthode Load de Forme (Rectangle, Pixel et Ligne) : **Load(const Liste<Couleur>& palette)**. Un parcours de cette liste à l'aide de l'itérateur permettra de retrouver la bonne couleur.

3.5 Importation de Couleurs et de Formes : Lecture de fichiers csv

Votre application devra être capable de permettre à l'utilisateur d'importer rapidement des couleurs et/ou formes à partir de fichiers csv. Ces **fichiers textes** peuvent être créés/lus à l'aide de Excel mais également dans n'importe quel éditeur de texte. Par exemple, pour les couleurs nous pourrions avoir

```
rouge;vert;bleu;nom
112;141;35;Olive
255;255;0;Jaune
253;108;158;Rose
206;206;206;Argent
```

La première ligne est l'**entête du fichier** et précise l'intitulé des champs de données. Elle n'est pas obligatoire mais serait indispensable si l'ordre des colonnes n'était pas figé. Chaque ligne représente donc une couleur et le caractère ';' est appelée le **séparateur**. Celui-ci pourrait être ':' ou encore ','.

On vous demande donc d'ajouter à la classe Dessin les méthodes suivantes :

- **importCouleurs(const char* nomFichier)** : qui importe les couleurs présentes dans le fichier csv dont le nom est passé en paramètre. A nouveau, il devra être impossible d'importer une couleur dont les composantes ne sont pas valables ou dont le nom est déjà présent dans la palette.
- **importRectangles(const char* nomFichier)** : qui importe un ensemble de Rectangles à partir du fichier csv dont le nom est passé en paramètre. Lors de la lecture du fichier, un identifiant unique sera attribué automatiquement au rectangle. Dans ce fichier csv, la couleur de chaque rectangle sera fournie à l'aide de son nom. Si un objet Couleur de ce nom n'est pas présent dans la palette, la forme ne sera pas importée.
- **importLigne(const char* nomFichier)** et **importPixels(const char* nomFichier)** : qui font la même chose que la méthode précédente mais pour des Lignes ou des Pixels.

Un ensemble de fichiers csv vous seront fournis au laboratoire.

3.6 La fenêtre graphique : La classe Dessin se dessine elle-même

Comme nous l'avons déjà mentionné, lorsque qu'un dessin est ouvert dans l'application, une fenêtre graphique est ouverte et affiche les différentes formes présentes dans le conteneur formes de la classe Dessin.

Pour gérer la fenêtre graphique, vous avez à votre disposition la classe **WindowSDL** qui 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.
- Pour ceux intéressés par le BONUS 😊 (voir plus loin) : Une 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.

Attention que ces différentes méthodes sont susceptibles de lancer l'exception **WindowSDLexception** que vous devrez gérer correctement (try... catch). Par exemple, si vous tentez d'ouvrir la fenêtre graphique alors qu'elle est déjà ouverte.

Au niveau de vos classes, on vous demande d'ajouter

- La méthode virtuelle pure **dessine()** à la classe **Forme**.
- Les méthodes **dessine()** aux classes héritées de **Forme**, c'est-à-dire **Ligne**, **Rectangle** et **Pixel**. Celles-ci devront appeler la méthode correspondante de la classe **WindowSDL**.

Remarquez que **ces méthodes dessine() ne devront pas ouvrir la fenêtre graphique !!!** Celle-ci est donc supposée ouverte au moment du dessin.

On vous demande également d'ajouter à votre classe **Dessin** la méthode **dessine()** qui :

1. Dessine le fond (utilisation de setBackground de WindowSDL)
2. Parcoure, à l'aide de l'itérateur, le conteneur forme de votre classe **Dessin** afin d'appeler la méthode **dessine()** de chacune des formes. Ce conteneur étant trié sur la profondeur des formes, l'affichage sera donc correct.

A nouveau, **cette méthode ne devra pas ouvrir la fenêtre graphique !!!** C'est donc votre application qui devra ouvrir/fermer la fenêtre graphique en fonction des actions dans le menu.

3.7 Fonctionnalités du menu : Description détaillée

Au démarrage de l'application, celle-ci ne contient aucune image. Elle ne devra être capable de gérer qu'un seul dessin à la fois, référencé par **un pointeur de type Dessin***. Au démarrage de l'application, ce pointeur est donc NULL.

Nous pouvons à présent aborder les différentes fonctionnalités du menu (voir section 3.1 de cet énoncé) :

a) Gestion de l'objet dessin et de la fenêtre graphique

- L'**item 0** permet simplement de quitter l'application proprement.
- L'**item 1** permet de créer un nouveau dessin (**allocation dynamique d'un objet Dessin**) dont les dimensions (**largeur** et **hauteur**) sont demandées à l'utilisateur. Ceci ne peut se faire que si un dessin n'est pas déjà ouvert. L'utilisateur devra donc fermer le dessin courant avant de pouvoir créer un nouveau. Une fois l'objet Dessin instancié, la fenêtre graphique doit s'ouvrir.
- L'**item 2** permet d'ouvrir un dessin à partir d'un fichier binaire d'extension « .des » dont le nom est demandé à l'utilisateur. Il suffit donc de faire appel à la méthode **Load** de la classe Dessin. De nouveau, ceci ne peut se faire que si un dessin n'est pas déjà ouvert. L'utilisateur devra donc fermer le dessin courant avant de pouvoir ouvrir un autre. Une fois l'objet Dessin instancié et chargé en mémoire à partir du fichier, la fenêtre graphique doit s'ouvrir et afficher le dessin qui vient d'être ouvert.
- L'**item 3** permet d'enregistrer le dessin courant dans un fichier binaire d'extension « .des » dont le nom est demandé à l'utilisateur. Il suffit pour cela de faire appel à la méthode **Save** de la classe Dessin.
- L'**item 4** permet de fermer proprement un dessin, c'est-à-dire de désallouer l'objet Dessin en cours (**delete**) et de fermer la fenêtre graphique.

La présence de la fenêtre graphique nous indique donc si un dessin est ouvert (en cours de modification) ou non.

b) Gestion générale

- L'**item 5** permet d'afficher les paramètres du dessin en cours, c'est-à-dire sa largeur, sa hauteur et sa couleur de fond.
- L'**item 6** permet à l'utilisateur de modifier la couleur de fond du dessin. Remarquez que lorsque la variable membre **fond** de l'objet Dessin a été modifiée, il faut appeler la méthode **dessine()** afin de mettre jour à la fenêtre graphique.
- L'**item 7** permet d'afficher le nombre d'objets Couleur, Point et Forme présents en mémoire à cet instant. Il suffit dès lors d'afficher les différentes variables membres statiques **Compteur** présentes dans ces classes.

b) Gestion de la palette de couleurs

- L'**item 8** permet d'ajouter, à la palette de couleurs du dessin, une nouvelle couleur dont les composantes rouge, vert, bleu et le nom sont demandés à l'utilisateur. Il suffit pour cela d'appeler la méthode **ajouteCouleur()** de l'objet Dessin. N'oubliez pas de **gérer les exceptions** ! En effet, il doit être impossible d'ajouter une couleur dont le nom est déjà présent dans la palette.
- L'**item 9** permet de modifier une couleur de la palette dont le nom est demandé à l'utilisateur. Il s'agira de modifier l'objet Couleur présent dans le conteneur palette (soit en demandant à l'utilisateur de ré-encoder les nouvelles composantes, soit en éclaircissant/assombrissant la couleur existante → utilisation des opérateurs + ou ++ ou – ou – de la classe Couleur). Tous les objets Forme pointant vers cet objet Couleur verront donc leur couleur modifiée. N'oubliez pas d'appeler la méthode **dessine()** de Dessin afin de mettre à jour la fenêtre graphique.
- L'**item 10** permet de supprimer, de la palette de couleurs, la couleur dont le nom est demandé à l'utilisateur. **Attention !** On ne pourra supprimer une couleur que si aucun objet Forme ne pointe dessus → la classe Dessin devra dans le cas contraire lancer une exception que vous devrez gérer !
- L'**item 11** permet d'afficher toutes les couleurs de la palette dans le terminal.
- L'**item 12** permet d'importer, dans le dessin courant, un ensemble de couleurs provenant d'un fichier csv dont le nom est demandé à l'utilisateur. Il suffit d'appeler la méthode **importCouleurs()** de la classe Dessin.

c) Gestion des formes géométriques

- L'**item 13** permet d'ajouter au dessin une forme dont toutes les caractéristiques sont demandées à l'utilisateur. Il faudra donc appeler la méthode **ajouteForme()** de la classe Dessin sans oublier de gérer correctement les exceptions.
- L'**item 14** permet de supprimer du dessin une forme dont l'identifiant est demandé à l'utilisateur. Remarquez que la couleur de cette forme ne sera pas supprimée.
- L'**item 15** permet de modifier la couleur d'une forme dont l'identifiant est demandé à l'utilisateur. Celui-ci devra choisir la nouvelle couleur de la forme parmi celles présentes dans la palette en encodant son nom.
- L'**item 16** permet d'afficher dans le terminal toutes les caractéristiques d'une forme dont l'identifiant est demandé à l'utilisateur.
- L'**item 17** permet d'importer un ensemble de pixels ou lignes ou rectangles à partir d'un fichier csv dont le nom est demandé à l'utilisateur. Remarquez qu'un fichier csv ne peut contenir qu'un seul type de forme.

Vous pouvez (devez ☺) bien évidemment ajouter à la classe Dessin toutes les méthodes nécessaires aux fonctionnalités évoquées dans le menu. N'oubliez pas qu'à chaque fois qu'une modification sera réalisée sur un des conteneurs de la classe Dessin, il sera nécessaire d'appeler la méthode **dessine()** de cette même classe afin de mettre à jour la fenêtre graphique.

3.8 **BONUS (non obligatoire donc) : Autres formes et Gestion du clic**

Bonus 1 : Ajout d'autres formes

Libre à vous de concevoir de nouvelles formes géométriques, comme des polygones ou des cercles. Il suffirait pour cela de définir de nouvelles classes ayant leurs propres variables membres (propres à la nouvelle forme → par exemple un rayon pour un cercle 😊) héritant de *Forme* en redéfinissant toutes ses méthodes virtuelles (pures).

Bonus 2 : Gestion du clic de la souris dans la fenêtre graphique

Comme nous l'avons déjà vu plus haut, la classe *WindowSDL* gérant la fenêtre graphique dispose de la méthode statique *waitClick()*, **bloquante**, permettant au programme de se mettre en attente jusqu'au moment où l'utilisateur aura cliqué quelque part dans la fenêtre graphique. Une fois cliqué, la méthode retourne un objet *WindowSDLclick* contenant les coordonnées (x,y) du clic. Ceci pourrait être utilisé afin de rendre l'ajout/suppression de forme plus sympathique 😊 :

- **Item 13** : lors de l'ajout d'une forme, au lieu d'encoder au clavier les coordonnées des points, il suffirait de les obtenir en cliquant dans la fenêtre graphique.
- **Item 14** : lors de la suppression d'une forme, au lieu d'encoder l'identifiant de la forme au clavier, il suffirait de cliquer quelque part dans la fenêtre graphique et la forme située « le plus près » du clic serait supprimée. Il faudra alors se baser sur un calcul de distance. Pour cela, il suffit de calculer la distance entre le clic et la position de la forme (variable membre **position** de la classe). Par exemple, pour supprimer un rectangle, il faudrait cliquer le plus proche possible de son coin supérieur gauche.
- **Item 15** : on modifierait la couleur de la forme sur laquelle l'utilisateur aurait cliqué.
- **Item 16** : on afficherait dans le terminal toutes les caractéristiques de la forme sur laquelle l'utilisateur aurait cliqué.

3.9 Fichiers csv fournis

Les fichiers *couleurs.csv*, *rectangles.csv*, *lignes.csv*, *pixels.csv* sont disponibles dans le centre de ressources de Jean-Marc Wagner. Au départ, il s'agit de fichiers Excel qui ont été enregistrés au format txt. Ces fichiers peuvent s'éditer avec Excel mais également avec "Notepad" ou n'importe quel éditeur de texte.