

## **Laboratoire de Programmation en C++**

**2<sup>ème</sup> informatique et systèmes :  
option(s) industrielle et réseaux (1<sup>er</sup> quadrimestre)  
et 2<sup>ème</sup> informatique de gestion (1<sup>er</sup> quadrimestre)**

**Année académique 2020-2021**

***Inpres Arts***

**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<sup>ème</sup> 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<sup>ème</sup> 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 2021 (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<sup>ère</sup> qu'en 2<sup>ème</sup> session.

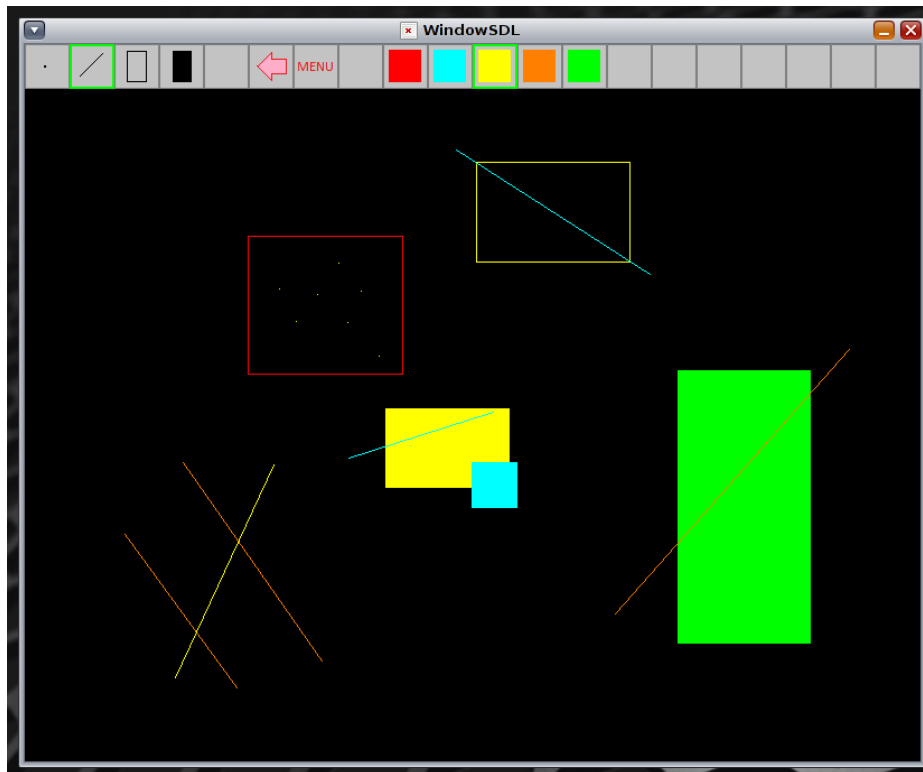
1) Chaque étudiant doit être capable d'expliquer et de justifier l'intégralité du travail.

2) En 2<sup>ème</sup> 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.

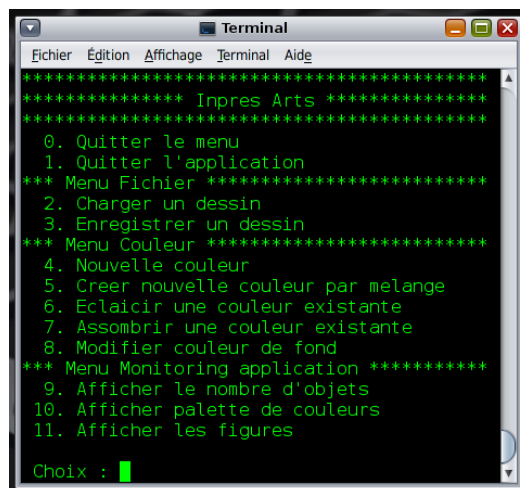
## 2. Le contexte : Inpres Arts

Les travaux de Programmation Orientée Objets (POO) C++ consistent à réaliser une application du type « Paint » (mais simplifiée !!). Dans un premier temps, il vous sera demandé de développer les éléments de base d'une telle application : point, pixel, couleur, figure, ligne, rectangle qui seront donc modélisés en C++ sous forme de classes. Une fois toutes ces entités mises au point, le but final est de développer une application graphique permettant de dessiner, à l'aide de la souris, un dessin composé de pixels, lignes, etc... Cette application aura l'aspect suivant :



Celle-ci présentera

- une zone de dessin,
- une barre d'outils permettant de sélectionner le type de tracé mais également la couleur,
- des boutons « undo » permettant de supprimer la dernière figure dessinée et un bouton « menu » faisant apparaître le menu suivant



### **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 proprement dite.

## 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 2021) :

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

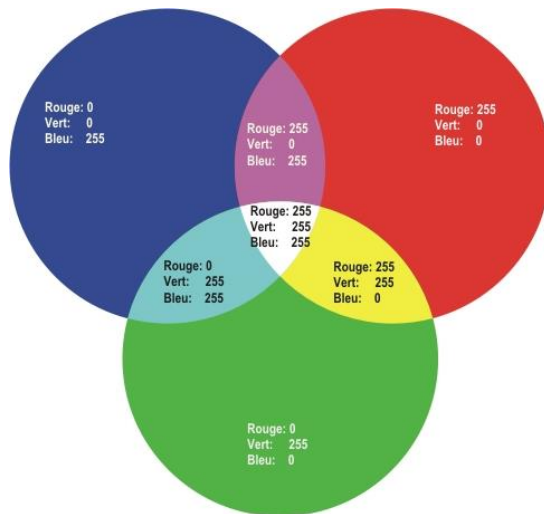
Etape	Thème	Page
1	Une première classe	6
2	Associations entre classes : agrégation + Variables statiques	7
3	Extension des classes existantes : surcharges des opérateurs	9
4	Associations de classes : héritage et virtualité	11
5	Les exceptions	13
6	Les containers et les templates	14
7	Première utilisation des flux	16
8	Un conteneur pour nos conteneurs : La classe Dessin	17
9	Enregistrement sur disque : La classe Dessin se sérialise elle-même	19
10	Mise en place de l'interface graphique : La classe ToolBar	20
11	Mise en place de l'application proprement dite	23
12	Mise en place du menu de l'application	24
13	Paramétrisation de l'interface graphique : lecture d'un fichier csv	26

**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 : Une première classe (Test1.cpp)

### 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.

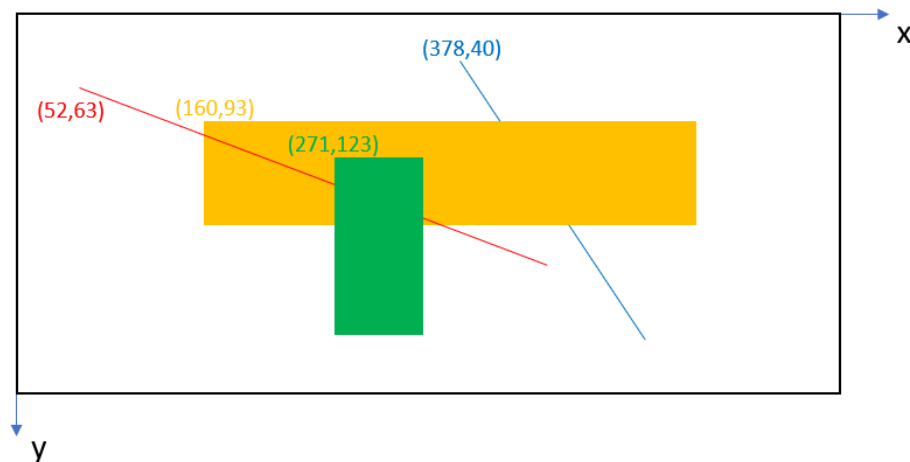
### b) Méthodologie de développement

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.

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

Il s'agit à présent de représenter en mémoire les objets graphiques ou plutôt les figures géométriques (ligne, rectangle, pixel, ...) que notre application sera capable de gérer. Tout d'abord, une figure géométrique est positionnée dans l'image, cette position sera caractérisée par ses coordonnées (x,y). Une forme possède également une couleur. Exemple :



Dans cet exemple, la ligne bleue a une position (1<sup>ère</sup> extrémité) située en (x,y) = (378,40) et a été dessinée avant le rectangle orange donc elle apparaît « en-dessous » de ce rectangle. Le rectangle vert plein a une position (coin en haut à gauche) égale à (x,y) = (271,123) et a été dessinée en dernier. L'ordre du dessin des figures géométriques n'est pas géré ici mais dans une étape ultérieure.

### a) La classe Point (Essai1())

Commençons par modéliser la notion de position d'une figure. 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 Figure (Essai2())

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

- Un **id** : une chaîne de 4 caractères (**char [5]**). 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.
- Deux constructeurs d'initialisation (voir jeu de tests). Une classe peut avoir plusieurs constructeurs d'initialisation. Par défaut ou par initialisation partielle, un objet Figure ne possède pas de couleur (NULL).
- Les méthodes getXXX()/setXXX() associées aux variables membres.
- La méthode Affiche() qui affiche toutes les caractéristiques de la figure géométrique.

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

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

La classe Figure 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 (ou simplement agrégation)**.

#### 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 Figure,

- 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. Celle méthode retourne la valeur du compteur.

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

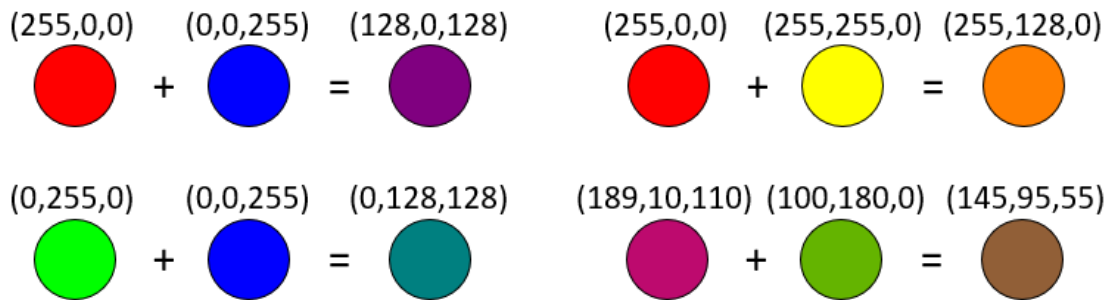
On se rend bien compte qu'il y a des couleurs qui apparaissent plus souvent que d'autres, comme le rouge, le vert ou le bleu. Dès lors, on pourrait imaginer de créer des **objets « permanents »** (dits « **statiques** ») existant même si le programmeur de l'application n'instancie aucun objet et représentant 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.



### Etape 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 faire la moyenne des composantes (rouge, vert, bleu) de chaque couleur. Exemples :



Par exemple, pour le mélange en bas à droite, nous avons

Rouge =  $(189 + 100) / 2 = 145$

Vert =  $(10 + 180) / 2 = 95$

Bleu =  $(110 + 0) / 2 = 55$

Remarquez que les composantes (rouge, vert, bleu) d'une couleur ne peuvent jamais dépasser la valeur maximale de 255. Heureusement, la moyenne de deux nombres compris entre 0 et 255 est toujours comprise entre 0 et 255.

#### 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 (178,122,31,"Melange de Carotte et de Olive")
```

Lorsque les deux couleurs possèdent un nom, le nom de la nouvelle couleur sera « Melange 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'éclaircir une couleur. 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 d'assombrir une couleur. 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 de l'opérateur [] de la classe Couleur : Essai5()**

Il s'agit ici de mettre en place l'opérateur [] de la classe Couleur permettant d'accéder directement aux composantes de la couleur et permettant d'exécuter un code du genre

```
Couleur c1(50,70,152);  
...  
cout << c1["rouge"] ;  
c1["vert"] = 110 ;
```

**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 les 3 composantes d'un objet Couleur **de 10** (tout en restant compris entre 0 et 255) et mettront à jour le nom. 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)
```

### **Etape 4 (Test4.cpp) :**

#### **Associations de classes : héritage et virtualité**

On se rend vite compte que la notion de figure géométrique est un peu limitée. En effet, une position seule ne suffit pas à décrire une figure géométrique. Nous allons donc *spécialiser* notre classe Figure afin d'obtenir des objets géométriques plus spécifiques comme les pixels, les lignes ou les rectangles. De plus, instancier la classe Figure 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 figure telle que nous l'avons imaginée. Toutes ces considérations nous mènent à concevoir la petite hiérarchie de classes décrite ci-dessous.

#### **a) La classe Figure 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 figures géométriques que nous considérerons. Cette classe de base sera évidemment notre classe **Figure** mais modifiée de la manière suivante :

- On lui ajoute la méthode virtuelle pure Dessine(), de type **void**, qui permettra, dans les classes héritées, de dessiner la figure géométrique dans une fenêtre graphique (voir plus bas). La classe Figure devient donc abstraite.

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

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

- possède ses propres constructeurs et l'opérateur << qui affiche une chaîne de caractères ayant le format suivant :

```
P12 [PIXEL : Position(10,40),Couleur(255,0,0,Rouge)]
```

pour le pixel d'identifiant P12, situé aux coordonnées (10,40) et de couleur rouge.

- redéfinit la méthode **Dessine()** qui dessine le pixel dans la fenêtre graphique. Pour cela, il faut utiliser la méthode `WindowSDL::drawPixel()` (une explication complète de la librairie `WindowSDL` est fournie à la fin de cet énoncé).

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

On vous demande de programmer la classe **Ligne**, qui hérite de la classe **Figure**, 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 l'opérateur `<<` qui affiche une chaîne de caractères ayant le format suivant :  

```
L17 [LIGNE : Position(30,90),Extremite(100,120),Couleur(0,255,0,Vert)]
```

pour la ligne d'identifiant L17 d'origine (30,90) d'extrémité (100,120) de couleur (0,255,0).
- redéfinit la méthode **Dessine()** qui dessine la ligne dans la fenêtre graphique. Pour cela, il faut utiliser la méthode `WindowSDL::drawLine()`

### d) La classe Rectangle : Essai3()

On vous demande de programmer la classe **Rectangle**, qui hérite de la classe **Figure**, 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.
- ses propres constructeurs et l'opérateur `<<` qui affiche une chaîne de caractères ayant le format suivant :  

```
R3 [RECTANGLE : Position(30,90),DimX=200,DimY=140,Rempli=1,Couleur(0,255,0,Vert)]
```

pour le rectangle d'identifiant R3, dont le coin supérieur gauche se situe en (30,90), de largeur 200, de hauteur 140, rempli, et de couleur (0,255,0).
- redéfinit la méthode **Dessine()** qui dessine le rectangle dans la fenêtre graphique. Pour cela, il faut utiliser la méthode `WindowSDL::fillRectangle()` si le rectangle est plein et `WindowSDL::drawRectangle()` sinon.

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

La méthode **Dessine()** 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++.

## Etape 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 Figure est invalide. Pour que l'identifiant d'un **Pixel** soit valide, il doit

- commencer par la lettre 'P'
- contenir 2, 3 ou 4 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 :
  - Un entier **valeur (int)** qui contiendra la valeur de la composante qui a posé problème.
  - Une méthode **Affiche()** qui affichera les caractéristiques de l'erreur. Exemple : « Composante rouge invalide ! (322) » où 322 est la valeur qu'on a essayé d'attribuer à la composante rouge de la couleur.

Par exemple, si c1 est un objet valide de la classe Couleur, c1.setRouge(300) lancera une InvalidColorException dont le message (hérité de BaseException) est « Composante rouge invalide ! » et dont la valeur est 300. Si maintenant, on tente de créer une couleur c2 en utilisant Couleur c2(110,300,100), ce constructeur lancera une InvalidColorException dont le message est « Composante verte invalide ! » et dont la valeur est 300. Si plusieurs composantes sont invalides simultanément, c'est la première trouvée qui génère l'exception.

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.

## Etape 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 figures géométriques d'un dessin 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) Une pile

On vous demande de programmer la **classe template Pile** qui hérite de la classe ListeBase qui redéfinit la méthode insere de telle sorte qu'un élément soit ajouté en début de liste chaînée, et qui dispose en plus :

- D'une méthode **push(...)** qui empile un élément sur la pile (elle ne fait qu'appeler la méthode insere déjà redéfinie).
- D'une méthode **top()** qui retourne l'élément situé au sommet de la pile mais sans altérer la pile.
- D'une méthode **pop()** qui retourne l'élément situé au sommet de la pile et qui le supprime de la pile.

Attention, les méthode top() et pop() retournent un objet template T et non une Cellule<T> !

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

### 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 pile mais nous n'avons aucun moyen de les parcourir, élément par élément. De plus, il est impossible de modifier ou supprimer un élément d'une liste. 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 Pile), et qui comporte, au minimum, les méthodes et opérateurs suivants :

- **reset()** qui réinitialise l'itérateur au début de la liste.
- **last()** qui positionne l'itérateur sur le dernier élément 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 - -** qui déplace l'itérateur vers la gauche.
- **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.

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 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 !!!)** et extrémité) et cela champ par champ. Le fichier obtenu sera un fichier **binaire** (utilisation des méthodes **write** et **read**).
- ♦ **Load(ifstream & fichier)** permettant de charger toutes les données relatives à une 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 (champ par champ) sur le flux fichier qu'il a reçu en paramètre.
- **void Load(ifstream & fichier)** : méthode permettant à un objet de se lire lui-même (champ par champ) 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.

## Etape 8 (Test8.cpp) :

### Un conteneur pour nos conteneurs : La classe Dessin

Un dessin est évidemment composé de *figures* 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 *couleur de fond* sur lequel les figures seront dessinées.

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

```
class Dessin
{
    private :
        Couleur fond;
        Liste<Couleur> palette;
        Pile<Figure*> figures;

    public :
        Dessin() ;
        Dessin(const Couleur& f) ;
        void ajouteCouleur(const Couleur& c) ;
        void ajouteFigure(Figure *pf,const char* nomCouleur) ;
        void Dessine();
        void undo();
        void AffichePalette();
        void AfficheFigures();
        ...
} ;
```

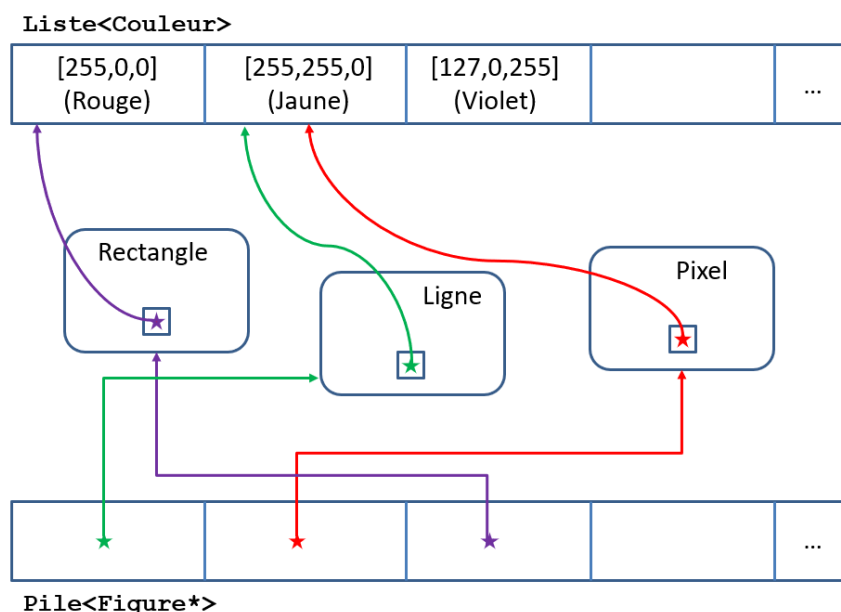
Cette classe comporte notamment :

- L'ensemble des données (variables membres) relatives au dessin.
- Des **constructeurs**.
- Une méthode **Couleur\*** **ajouteCouleur(const Couleur& c)** 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. La méthode retournera l'adresse de la couleur insérée dans le conteneur.

- Une méthode **ajouteFigure** qui permettra d'ajouter, au conteneur **figures**, une figure allouée dynamiquement avant l'appel de la méthode. Cet objet n'aura au départ aucune couleur. Le second paramètre de la méthode ajouteFigure permettra de lui attribuer une couleur faisant partie de la palette (la couleur devra donc déjà exister !). L'objet Figure 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.
- Une méthode **Dessine()** qui permet de dessiner l'ensemble des figures dans la fenêtre graphique. Pour cela, le fond est tout d'abord dessiné (utilisation de WindowSDL::setBackground() pour l'instant). Ensuite, l'itérateur permettra de parcourir la pile du bas vers le sommet et d'appeler la méthode Dessine() de chaque forme. Attention qu'aucune méthode Dessine() ne doit ouvrir la fenêtre graphique, elles supposent donc que la fenêtre est déjà ouverte.
- Une méthode **undo()** qui permet de supprimer la dernière forme ajoutée. Pour cela, il suffit de dépiler la pile de son sommet.
- Une méthode **AffichePalette()** qui affiche le nombre de couleurs présentes dans la palette et affiche à l'aide de l'itérateur chacune des couleurs à l'aide de son opérateur <<.
- Une méthode **AfficheFigures()** qui affiche le nombre de figures présentes dans le conteneur figures et affiche à l'aide de l'itérateur chacune des figures à l'aide de son opérateur <<.
- Un **destructeur** qui balayera la pile de figures afin de désallouer correctement toutes les figures du dessin.
- ...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 figures 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 ligne est au sommet de la pile, c'est la dernière figure qui a été dessinée. La méthode Dessine() de la classe Dessin doit dessiner le rectangle, puis le pixel et enfin la ligne.

## Etape 9 :

### 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. Couleur de fond : l'objet **fond**
2. 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.
3. Les figures géométriques : pour cela, on écrira tout d'abord le **nombre de figures**, puis ensuite **chacun des objets « Figure »**. 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 figures lire et le caractère lu avant chaque figure permettra de savoir quel type de figure allouer.

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

- La méthode **Save(const char\* nomFichier)** qui enregistrera dans le fichier « nomFichier » 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, Figure, Ligne, ...). Chaque objet va donc se sérialiser lui-même.

### Plusieurs remarques importantes :

- Lors de l'**écriture d'une figure**, 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 figures**, le conteneur figures est parcouru à l'aide de l'itérateur. Mais comment savoir quel type d'objet est pointé par le pointeur Figure\* 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 Figure... Pourquoi 😊 ?

- Lors de la **lecture d'une figure**, 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 Figure (Rectangle, Pixel et Ligne) : **Load(ifstream& fichier, const Liste<Couleur>& palette)**. Un parcours de cette liste à l'aide de l'itérateur permettra de retrouver la bonne couleur.

## Etape 10 (Test10.cpp):

### Mise en place de l'interface graphique : **La classe ToolBar**

Il est temps de mettre en place les éléments permettant de construire un dessin à l'aide de l'interface graphique, et en particulier à l'aide de la barre d'outils apparaissant en haut de la fenêtre graphique. Plusieurs classes vont devoir être développées : Bouton, BoutonCouleur et ToolBar. En effet, la librairie WindowSDL permet d'importer une image de fond (qui vous sera fournie) et présente quelques primitives graphiques de base (pixel, ligne, etc) mais elle ne dispose pas de boutons ou de menu comme en java ou C#. Les classes que l'on va mettre en place vont permettre de palier à cet inconvénient.

On vous demande tout d'abord de développer la classe **Bouton** qui présente :

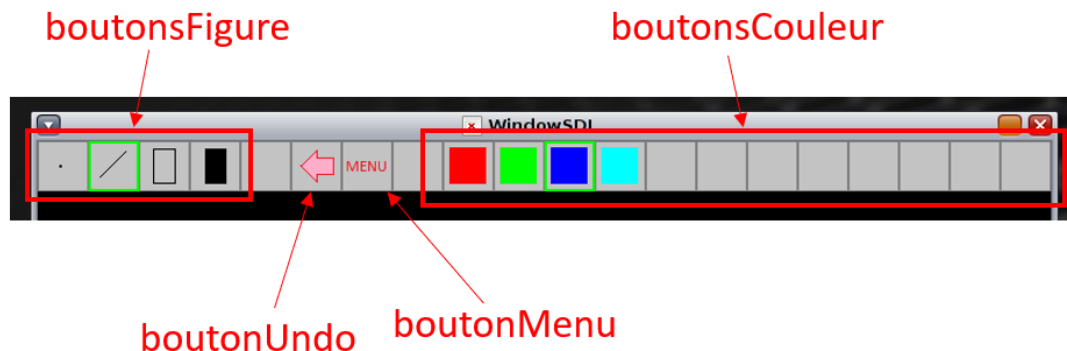
- Les variables **X1** et **X2** (de type **Point**) qui représentent respectivement le coin supérieur gauche et le coin inférieur droit du bouton)
- La variable **nom** (de type **char\***) qui représente le nom du bouton. Exemples : « B\_LIGNE », « B\_UNDO », ...
- La variable **selected** (de type **bool**) qui représente le fait que le bouton est sélectionné ou pas. Dans le cas où il est sélectionné, celui-ci sera encadré d'un rectangle vert (par exemple) afin de signifier à l'utilisateur qu'il est sélectionné.
- La méthode **bool clic(int x,int y)** qui retourne **true** si le clic matérialisé par le point de coordonnées (x,y) se fait sur le bouton, et **false** sinon. Il suffit pour cela de comparer x et y aux coordonnées des points X1 et X2.
- La méthode **void setSelected(bool s)** qui a bien évidemment le rôle de modifier la variable **selected** mais également de
  - Dessiner un cadre vert sur tout le contour du bouton (utilisation de `WindowSDL::drawRectangle()`) si s vaut **true**
  - Effacer ce cadre vert en dessinant un rectangle gris sur le contour du bouton si s vaut **false**. Pour information, les couleurs du bouton sont (127,127,127) et (195,195,195).
- Les constructeurs traditionnels, les getters/setters et les opérateurs nécessaires à l'utilisation de la classe Bouton dans les conteneurs développés plus haut.

La classe **Bouton** va nous permettre de représenter les 4 quatre boutons de gauche (outils pixel, ligne, rectangle creux et rectangle plein) et les 2 boutons « Menu » et « Undo ». Par contre, il nous faut une autre classe nous permettant de sélectionner une des couleurs de la palette.

Pour cela, on vous demande de développer la classe **BoutonCouleur** qui hérite de la classe **Bouton** et qui possède en plus

- Une variable **couleur** (pointeur de type **Couleur\***) qui pointera vers la couleur que ce bouton représente.
- Une méthode **void setCouleur(Couleur \*)** permettant de mettre à jour la variable couleur mais également de dessiner dans le bouton un petit rectangle (de dimension 10 de moins en largeur et en hauteur que le bouton) rempli de la couleur pointée (utilisation de `WindowSDL::fillRectangle()`).
- La surcharge de la méthode **void setSelected(bool s)** qui empêchera que l'on puisse sélectionner un **BoutonCouleur** non associé à une couleur (dans ce cas, le pointeur couleur est NULL).
- Les constructeurs traditionnels, les getters/setters et les opérateurs nécessaires à l'utilisation de la classe **BoutonCouleur** dans les conteneurs développés plus haut.

L'ensemble des boutons apparaissant en haut de la fenêtre graphique porte le nom de barre d'outils et comporte les éléments suivants :



- Les 4 boutons de gauche sont ceux qui permettent de choisir la figure ou « l'outil de dessin », c'est-à-dire « Pixel », « Ligne », « Rectangle creux » ou « Rectangle plein ». Lorsqu'un de ces 4 boutons est sélectionné, les autres ne peuvent pas l'être car un seul outil peut être sélectionné à la fois. Ces 4 boutons forment donc un « groupe de boutons » et seront placés dans un même conteneur « **boutonsFigure** ». Il s'agit d'instance de la classe **Bouton**.
- Les boutons « **Undo** » et « **Menu** » sont indépendants de tous les autres boutons et ne peuvent pas être sélectionnés. En effet, un clic sur l'un des deux ne sélectionne pas un outil (ou une couleur) mais réalise une action (voir plus loin). Néanmoins, il s'agit tout de même d'instances de la classe **Bouton** mais qui ne seront jamais sélectionnées.
- Les 12 boutons de droites permettent de sélectionner une couleur dans la palette de couleur du dessin en cours. Lorsqu'un de ces boutons est sélectionné, aucun des 11 autres ne peut l'être (rappelons-nous également qu'un **BoutonCouleur** non affecté à une couleur ne pourra pas être sélectionné). Ces 12 boutons sont donc des instances de la classe **BoutonCouleur**, forment un autre « groupe de boutons » et seront donc placés dans un même conteneur « **boutonsCouleur** ».

Etant donné ces explications, on vous demande de développer la **ToolBar** qui aura la structure suivante :

```
class ToolBar
{
    private :
        Liste<Bouton>          boutonsFigure;
        Liste<BoutonCouleur>   boutonsCouleur;
        Bouton                 boutonUndo;
        Bouton                 boutonMenu;

    public :
        ToolBar() ;
        ToolBar(const ToolBar& f);

        void ajouteBoutonFigure(const Bouton& b);
        void ajouteBoutonCouleur(const BoutonCouleur& b);
        void setBoutonUndo(const Bouton& b);
        void setBoutonMenu(const Bouton& b);

        void ajouteCouleur(Couleur* c);

        const char* clic(x,y);
} ;
```

Cette classe comporte notamment :

- Deux constructeurs
- Les méthodes d'ajout des différents boutons.
- Une méthode **void ajouteCouleur(Couleur\* c)** qui affecte la couleur c au premier **BoutonCouleur** de la liste **boutonsCouleur** dont le pointeur couleur est NULL. Pour cela, il suffit d'utiliser l'itérateur.
- Une méthode **const char\* clic(x,y)** qui « balaye » tous les boutons contenus dans la **ToolBar** (ceux présents dans les deux listes et les deux boutons indépendants) et teste si tel ou tel bouton a été cliqué en (x,y). De plus, cette méthode sélectionne/désélectionne les boutons par groupe et assure donc qu'un seul outil et une seule couleur peuvent être sélectionnés à la fois. Il suffit pour cela d'utiliser (sans modération) l'itérateur. La méthode retournera alors
  - o soit le nom du bouton qui a été cliqué,
  - o soit NULL si aucun bouton n'a été cliqué.
- Une méthode **Couleur\* getCouleur(const char\* nomBoutonCouleur)** qui retourne un pointeur vers la couleur associée au **BoutonCouleur** dont le nom est passé en paramètre à la méthode.
- Une méthode **Couleur\* getCouleurSelectionnee()** qui retourne un pointeur vers la couleur associée au BoutonCouleur sélectionné.
- Une méthode **const char\* getFigureSelectionnee()** qui retourne le nom du bouton « Figure » sélectionné (« B\_PIXEL », « B\_LIGNE », ...)

Le fichier Test10.cpp vous aidera à mettre au point ces différentes classes. Il fournit également les caractéristiques (nom, coordonnées des points) des différents boutons de l'application, ceux-ci étant directement liés à l'image de fond également fournie.

## Etape 11 (Test11.cpp) :

### Mise en place de l'application proprement dite

Cette étape est très courte car elle ne nécessite aucune programmation de votre part. Le fichier « Test11.cpp » constitue la base du « **main** » de votre application finale. Le but ici est donc de comprendre comment il est structuré afin de poursuivre le développement. Voici néanmoins quelques remarques importantes :

- Il instancie un objet de la classe **ToolBar** et un objet de la classe **Dessin** et fait le lien entre le deux.
- Il contient les pointeurs **figure** (de type **const char\***) et **couleur** (de type **Couleur\***) qui pointent respectivement vers le nom de l'outil graphique en cours (« B\_PIXEL », « B\_LIGNE », ... ou NULL si aucune figure n'a encore été sélectionnée par l'utilisateur) et vers la couleur sélectionnée (ou NULL si aucune couleur n'a encore été sélectionnée par l'utilisateur).
- Il contient une boucle infinie que l'on appelle la « boucle des événements » en programmation graphique. Dès qu'un clic sur la fenêtre graphique est détecté, l'objet **ToolBar** permet de savoir si on a cliqué sur un bouton ou sur la zone de dessin et d'agir en conséquence.
- Actuellement les trois couleurs de base (rouge, vert, bleu) ont été ajoutées « en dur » au dessin ainsi qu'à la barre d'outils. Ceci disparaîtra dans la suite lorsque le menu de l'application sera mis en place (voir plus loin). Celui-ci permettra notamment à l'utilisateur d'encoder ses propres couleurs.
- Les identifiants des figures ne sont pas gérés correctement. A vous de générer automatiquement des identifiants de figure qui respectent les contraintes. Pour cela, vous pouvez par exemple utiliser un petit compteur statique, variable membre de la classe **Figure** (attention que cela implique que les méthodes Save() et Load() de Figure soient mises à jour afin d'en tenir compte).

## Etape 12 :

### Mise en place du menu de l'application

L'application peut se trouver dans 2 modes différents :

- Soit en **mode graphique**, c'est-à-dire que le contrôle est géré par la fenêtre graphique et l'application ne réagit qu'à un clic dans cette fenêtre. La console ne sert dans ce cas qu'à afficher des traces de l'application et des informations pour l'utilisateur.
- Soit en **mode console**, c'est ce qu'il se produit lorsque l'on clique sur le bouton « Menu ». Ce clic fait apparaître un menu dans la console et le contrôle est alors géré par la console par l'intermédiaire du clavier. En mode console, un clic sur la fenêtre graphique n'a aucun effet. Une fois sorti de ce menu, on se retrouve de nouveau en mode graphique.

Lorsque l'on clique sur le bouton « Menu » voici donc le menu qui apparaît dans la console :

```
*****
***** Inpres Arts *****
*****
0. Quitter le menu
1. Quitter l'application
*** Menu Fichier *****
2. Charger un dessin
3. Enregistrer un dessin
*** Menu Couleur *****
4. Nouvelle couleur
5. Creer nouvelle couleur par melange
6. Eclaircir une couleur existante
7. Assombrir une couleur existante
8. Modifier couleur de fond
*** Menu Monitoring application *****
9. Afficher le nombre d'objets
10. Afficher palette de couleurs
11. Afficher les figures

Choix : 
```

Passons en revue les différents items de ce menu :

- L'**item 0** permet simplement de quitter le mode console (et donc le menu) et de revenir au mode graphique.
- L'**item 1** ferme la fenêtre graphique et termine proprement l'application (exactement comme le fait le clic sur la croix de la fenêtre en mode graphique)
- Les **items 2 et 3** permettent d'enregistrer/charger un dessin vers/à partir d'un fichier binaire sur disque. Après avoir demandé le nom du fichier à l'utilisateur, il suffit d'utiliser les méthodes Save() et Load() de la classe Dessin. Au moment du chargement, il est également nécessaire de mettre à jour les **BoutonCouleur** de la **ToolBar**. Pour cela, ... soyez imagitatif !
- L'**item 4** permet de créer une nouvelle couleur en demandant à l'utilisateur d'encoder ses composantes et son nom. Une fois l'objet Couleur instancié, il sera inséré dans le dessin mais également pointé par un des BoutonCouleur. Attention que le BoutonCouleur en



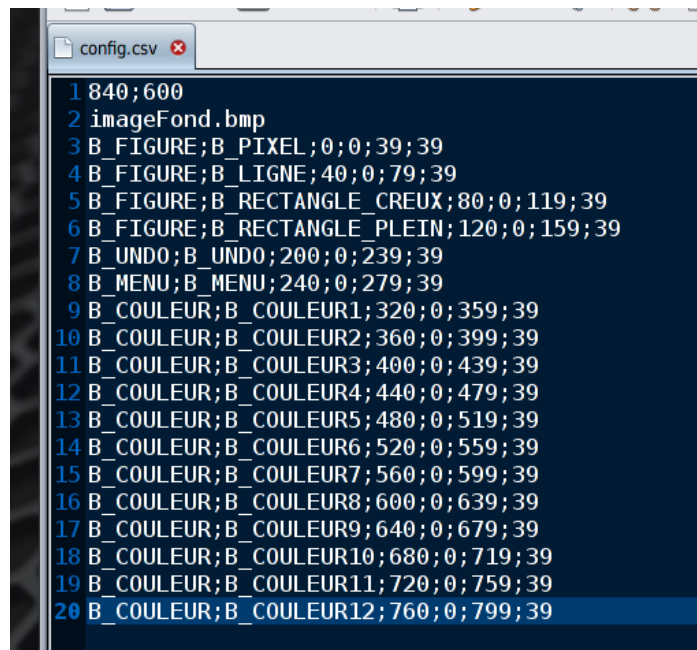
question pointera vers l'objet situé dans le conteneur palette de l'objet Dessin et non vers une copie de cette couleur ! Enfin, attention à bien gérer l'**exception** générée par l'opérateur >> de la classe Couleur.

- L'**item 5** permet de créer une nouvelle couleur en utilisant l'opérateur « Couleur + Couleur » de la classe Couleur. On demande dans un premier temps à l'utilisateur de sélectionner deux couleurs en cliquant successivement les BoutonCouleur correspondants. Ensuite, vous devez utiliser les méthodes WindowSDL::clic(), const char\* clic(int x,int y) de la classe ToolBar mais également Couleur\* getCouleur(const char\* nomBoutonCouleur) de la classe ToolBar. Cela aura ensuite pour effet de créer et d'ajouter une nouvelle couleur à la palette du dessin et à la ToolBar.
- L'**item 6** permet d'éclaircir une couleur existante sans en créer une nouvelle. On demande tout d'abord à l'utilisateur de cliquer sur une des couleurs de la palette (dans la ToolBar). Ensuite, on lui demande de combien on veut éclaircir cette couleur (int). On applique alors l'opérateur « Couleur + int » à la couleur sélectionnée. Attention que cette modification doit se répercuter sur la ToolBar mais également sur toutes les figures du dessin qui pointent vers cette couleur !
- L'**item 7** est similaire à l'item 6 à la différence que l'on utilise l'opérateur « Couleur – int ».
- L'**item 8** demande à l'utilisateur de saisir une nouvelle couleur (attention aux exceptions !). Celle-ci sera alors attribuée à la couleur de fond du dessin en cours qui devra donc être redessiné.
- L'**item 9** affiche le nombre d'objets de type Point, Couleur et Figure. Il suffit pour cela d'afficher les compteurs statiques qui ont été mis en place au début du projet.
- L'**Item 10** appelle simplement la méthode AffichePalette() de l'objet dessin.
- L'**Item 11** appelle simplement la méthode AfficheFigures() de l'objet dessin.

## Etape 13 :

### Paramétrisation de l'interface graphique : lecture d'un fichier csv

Jusqu'à maintenant, les caractéristiques des boutons (nom, position) étaient codées en dur dans l'application. Or, ces boutons dépendent également de l'image de fond qui a été fournie. Si on décide de changer cette image de fond, la position ou le nombre de boutons, cela ne correspond plus au code actuel. Il serait judicieux que l'application lise, au démarrage, un fichier texte contenant toutes ces informations. On vous fournira donc le fichier text « config.csv » qui peut être lu et édité à l'aide de n'importe quel éditeur. Voici ce fichier :



```
1 840;600
2 imageFond.bmp
3 B_FIGURE;B_PIXEL;0;0;39;39
4 B_FIGURE;B_LIGNE;40;0;79;39
5 B_FIGURE;B_RECTANGLE_CREUX;80;0;119;39
6 B_FIGURE;B_RECTANGLE_PLEIN;120;0;159;39
7 B_UNDO;B_UNDO;200;0;239;39
8 B_MENU;B_MENU;240;0;279;39
9 B_COULEUR;B_COULEUR1;320;0;359;39
10 B_COULEUR;B_COULEUR2;360;0;399;39
11 B_COULEUR;B_COULEUR3;400;0;439;39
12 B_COULEUR;B_COULEUR4;440;0;479;39
13 B_COULEUR;B_COULEUR5;480;0;519;39
14 B_COULEUR;B_COULEUR6;520;0;559;39
15 B_COULEUR;B_COULEUR7;560;0;599;39
16 B_COULEUR;B_COULEUR8;600;0;639;39
17 B_COULEUR;B_COULEUR9;640;0;679;39
18 B_COULEUR;B_COULEUR10;680;0;719;39
19 B_COULEUR;B_COULEUR11;720;0;759;39
20 B_COULEUR;B_COULEUR12;760;0;799;39
```

Il s'agit d'un fichier csv. Les deux premières lignes constituent l'*entête du fichier* et précisent la taille (1<sup>ère</sup> ligne) et le nom du fichier (2<sup>ème</sup> ligne) de l'image de fond à charger. Ensuite, chaque ligne représente un bouton et le caractère ';' est appelée le *séparateur*. Chaque ligne est structurée de la manière suivante : « type;nom;x1;y1;x2;y2 » où

- type permet de savoir à quel type de bouton nous avons affaire
- nom correspond au nom du bouton
- (x1,y1) correspond au coin supérieur gauche du bouton
- (x2,y2) correspond au coin inférieur droit du bouton

Le but ici est de modifier la méthode void InitGUI() préalablement fournie pour qu'elles tiennent compte des informations présentes dans le fichier csv.

## **BONUS (non obligatoire donc)**

### **Bonus 1 : Suppression de couleurs de la palette**

Rien n'a été dit concernant l'éventuel suppression d'une couleur de la palette d'un dessin. Il s'agit de mettre en place les éléments nécessaires pour supprimer une couleur de la palette d'un dessin mais également de la ToolBar. Attention qu'une couleur ne peut être supprimée que si aucune figure ne pointe dessus.

### **Bonus 2 : Ajout d'autres figures**

Libre à vous de concevoir de nouvelles figures 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 figure → par exemple un rayon pour un cercle 😊) héritant de Figure en redéfinissant ses méthodes virtuelles (pures). De plus, l'image de fond devrait être modifiée pour tenir compte de nouveaux boutons et, de là, le fichier « config.csv » devrait également être mis à jour.

Bon travail !

## Librairie WindowSDL

Afin de gérer l'aspect graphique de l'application, 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 (dont celles en rouge vous intéressent particulièrement):

- **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.
- **void drawPixel(int r,int v,int b,int x,int y)** qui permet d'afficher un pixel (x,y) de couleur (r,v,b)
- **void drawLine(int r,int v,int b,int x1,int y1,int x2,int y2)** qui permet d'afficher une ligne d'origine (x1,y1), d'extrémité (x2,y2) et de couleur (r,v,b)
- **void drawRectangle(int r,int v,int b,int x,int y,int w,int h)** qui permet d'afficher un rectangle creux de coin supérieur gauche (x,y), de largeur w, de hauteur h et de couleur (r,v,b)
- **void fillRectangle(int r,int v,int b,int x,int y,int w,int h)** qui permet d'afficher un rectangle plein de coin supérieur gauche (x,y), de largeur w, de hauteur h et de couleur (r,v,b)
- **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.