Laboratoire de Programmation en C++

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

Année académique 2018-2019

Calculette

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) 2ème 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) 2ème 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 2017 (sur base d'une liste de points de théorie fournis 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 1ère qu'en 2ème session.

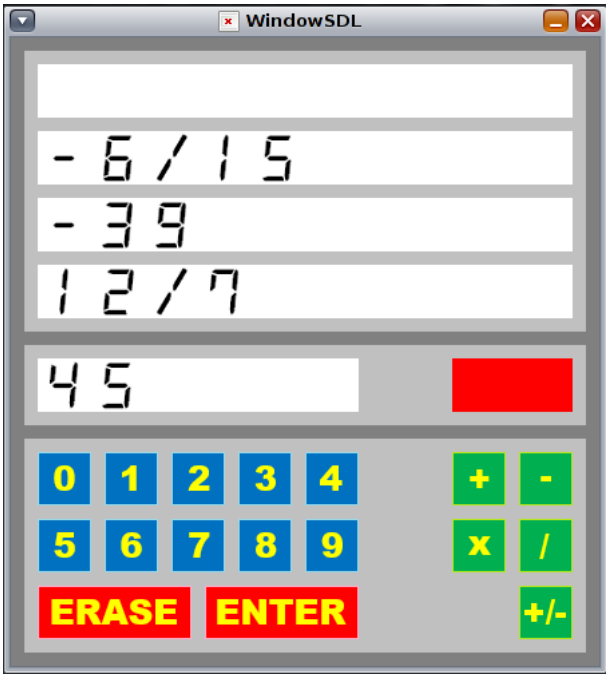
1) Chacun des membres d'une équipe d'étudiants doit être capable d'expliquer et de justifier l'intégralité du travail (pas seulement les parties du travail sur lesquelles il aurait plus particulièrement travaillé)

2) En 2ème session, un **report de note** est possible pour la théorie ou le laboratoire **pour des notes supérieures ou égales à 10/20**. Les évaluations (théorie ou laboratoire) ayant des **notes inférieures à 10/20** sont **à représenter dans leur intégralité**. En seconde session, **seul comptera l'évaluation du projet final pour la cote de laboratoire**.

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 : Calculette

Les travaux de Programmation Orientée Objets (POO) C++ consistent à réaliser une calculatrice (graphique) permettant de manipuler des nombres fractionnaires (des fractions). Voici à quoi devrait ressembler votre calculatrice au terme du laboratoire :



Cette calculatrice comporte 4 lignes d'affichage permettant de stocker des résultats de calculs intermédiaires (vide, -6/15, -39, 12/7), et une zone de saisie (45). Pour la saisie de nombres et les calculs proprement dit, la calculette dispose de différents boutons (bleus pour les chiffres, verts pour les opérations et rouges pour des opérations spéciales : « erase » vide toutes les zones d'affichage et de saisie et « enter » permet de terminer la saisie d'un nombre. Comme on le verra plus tard, il s'agit d'une calculatrice utilisant la notation polonaise inverse (NPI). Plus d'explications seront fournies dans la seconde partie de l'énoncé.

Il s'agit donc ici de mettre en place dans un premier temps les bases fonctionnelles d'une telle application. Nous verrons donc apparaître les classes Fraction, Bouton, Panneau (zone rectangulaire de couleur uniforme ; par exemple les trois rectangles gris clairs apparaissant dans la fenêtre graphique), BoutonImage (des boutons comportant des images comme le bouton « enter »), PanneauImage (des panneaux comportant des images comme les affichages des chiffres dans les 4 lignes d'affichage et la zone de saisie de la calculatrice.

Dans la première partie de l'énoncé, toutes les classes énoncées plus haut seront développées sans l'utilisation de fenêtre graphique. A terme (seconde partie de l'énoncé), il sera néanmoins possible d'ouvrir une fenêtre graphique comme celle montrée ci-dessus, à l'aide d'une mini-librairie C++ graphique que l'on vous fournira.

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 1er quart puis de conforter vos acquis au 2ème 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 élaborez 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 mi-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.

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

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

0.5 Planning et contenu des évaluations

a) Evaluation 1 (continue) :

Porte sur : les 7 premiers jeux de tests (voir tableau donné plus loin).

Date de remise du dossier (dépôt sur la machine Sunray de l'école) : un lundi (12h00 au plus tard) entre la mi-novembre et fin novembre (la date sera fixée précisément et vous sera fournie par vos professeurs de C++).

Modalités d'évaluation : à partir de la date de remise du dossier selon les modalités indiquées par le professeur de laboratoire.

b) Evaluation 2 (examen de janvier 2019) :

Porte sur : le développement de l'application finale (voir seconde partie de l'énoncé).

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)

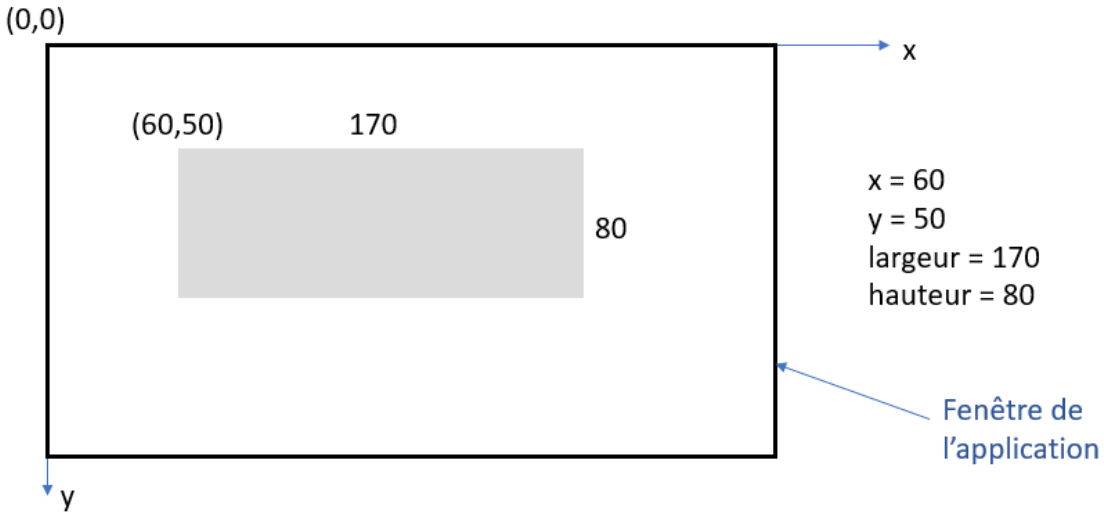
Points principaux de l'évaluation	Subdivisions
• **Jeu de tests 1** : Implémentation d'une classe de base (**Panneau)**	Constructeurs, destructeurs
	Getters, Setters et méthode Affiche
	Fichiers Panneau.cpp, Panneau.h et makefile
• **Jeu de tests 2 :** Agrégation entre classes (classe **Couleur**)	La classe **Couleur**
	Agrégation par valeur (classe **Panneau** et **Couleur**)
	Variables membres statiques (type Couleur)
• **Jeu de tests 3 :** Surcharge des opérateurs	Operateurs =, << et >> des classes Couleur et Panneau
	La classe **Fraction**
	Variables (UN et ZERO) et méthode (PGCD) **statiques**
	Opérateurs = et <<
	Opérateurs +, -, *, /
	Opérateur ++
• **Jeu de tests 4 :** Héritage et virtualité	Classes **Bouton** et **PanneauImage** : héritage simple
	Classe **BoutonImage** : héritage multiple
	Test de la virtualité et du down-casting
• **Jeu de tests 5 :** Exceptions	**BaseException**
	FractionException qui hérite de BaseException
	Utilisation correcte de **try**, **catch** et **throw**
• **Jeu de test 6 :** Containers génériques	Classe **abstraite ListeBase**
	Classe **Liste** (→ int et PanneauImage)
	Classe **ListeTriee** (→ int et Fraction)
	Itérateur : classe **Iterateur**
	BONUS : classe **Pile** (→ int et Fraction)
• **Jeu de test 7 :** Flux	Méthodes **Save()** et **Load()** de Couleur
	Méthodes **Save()** et **Load()** de Panneau
	Méthodes **Save()** et **Load()** de PanneauImage

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 graphique est la notion de panneau. Un panneau est une zone rectangulaire de la fenêtre graphique, caractérisée par les coordonnées (x,y) de sa position (coin supérieur gauche de la zone rectangulaire) et ses dimensions (largeur, hauteur). Voici un exemple :



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

- **x**, **y** : deux entiers (**int**) représentant sa position (en pixels) dans la fenêtre graphique.
- **largeur, hauteur** : deux entiers (**int**) représentant les dimensions (en pixels) du panneau.
- Un **nom** : une chaine de caractères allouée dynamiquement (**char ***) en fonction du texte qui lui est associé. Il s'agit du nom du panneau qui permettra de l'identifier par la suite.

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 Panneau (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 : agrégation + Variables statiques

a) La classe Couleur (+ variables membres statiques)

Un panneau possède en plus une certaine couleur. Dès lors, nous allons compléter notre classe Panneau. Mais avant cela, on vous demande de développer la classe **Couleur**, qui est caractérisée par :

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

Différents constructeurs sont demandés, comme vous pourrez le voir dans le jeu de Test2.cpp. La couleur construite par le constructeur par défaut sera le noir (nom="Noir", rouge=vert=bleu=0). Le constructeur n'ayant qu'un seul paramètre entier (pour le gris) crée une couleur grise dont le nom sera obtenu par la concaténation de "Gris" et de la valeur entière reçue en paramètre.

On se rend bien compte qu'il y a des couleurs qui apparaissent plus souvent que d'autres, comme le gris 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 BLEU, GRIS et ARGENT, statiques, constantes de type Couleur et ayant les caractéristiques respectives (r=0,v=0,b=255,nom="Bleu"), (r=128,v=128,b=128,nom="Gris") et (r=192,v=192,b=192,nom="Argent"). Voir jeu de tests.

a) Modification de la classe Panneau (agrégation par valeur)

Afin de compléter la classe Panneau, on vous demande de lui ajouter une nouvelle variable membre appelée **couleur** dont le type est la classe **Couleur**. De plus,

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

1.3 Jeu de tests 3 (Test3.cpp) : Surcharges des opérateurs

Il s'agit ici de surcharger un certain nombre d'opérateurs des classes développées ci-dessus afin d'en étendre les fonctionnalités. Une nouvelle classe (Fraction) sera ensuite développée.

a) Surcharge des opérateurs =, << et >> des classes Couleur et Panneau (Essai1() à Essai4())

La première chose à faire est de développer les opérateurs =, << et >> des classes Couleur et Panneau afin qu'elles soient « complètes » (concept d'abstraction). Par exemple, la compilation et l'exécution de ce genre de code doit être possible :

```
Couleur c1(112,141,35,"Argent"), c, c2, c3 ;
c = c1 ;
c2 = c3 = Couleur(145) ;
cin >> c2 ;
cout << c3 << endl ;

Panneau p1("P01",20,30,300,180,Couleur(158)), p, p2, p3;
p = p1 ;
p2 = p3 = Panneau("P02",30,30,400,300) ;
cin >> p3 ;
cout << p3 << endl ;
```

b) La classe Fraction (Essai5())

L'élément essentiel de notre future calculatrice est le type de nombre qu'elle va manipuler. Dans notre cas, il s'agira de nombres fractionnaires, ou plus simplement de fractions. On vous demande donc tout d'abord de développer la classe **Fraction** qui possède les variables membres suivantes :

- **numerateur** : un entier positif (**unsigned int**) qui représente le numérateur de la fraction.
- **denominateur** : un entier positif (**unsigned int**) qui représente le dénominateur de la fraction.
- **signe** : une variable de type **Signe** ne pouvant prendre que les valeurs +1 ou -1 et qui représente le signe de la fraction.

En réalité, Signe est un **enum** que vous devez déclarer ainsi (en public) dans la classe Fraction :

```
enum Signe { positif = +1, negatif = -1 } ;
```

Dès lors, la variable membre **signe** de la classe Fraction ne pourra prendre que les valeurs **positif (+1)** ou **negatif (-1)**.

La classe Fraction va également disposer des **constructeurs** suivants :

- **Fraction()** : crée une fraction égale à 1 (numerateur = 1, denominateur = 1, signe = positif)
- **Fraction(unsigned int n, unsigned int d, Signe s)** : crée une fraction de numerateur = n, denominateur = d et signe = s (n et d doivent être positifs sous peine d'erreur…).
- **Fraction(int n, int d)** : crée une fraction de numerateur = abs(n), denominateur = abs(d) et signe = le signe du produit de n et de d. Exemples : Fraction(2,-3) donne numerateur = 2, denominateur = 3 et signe = negatif. Fraction(-1,-5) donne numerateur = 1, denominateur = 5 et signe = positif.
- **Fraction(int n)** : crée un nombre entier de numerateur = abs(n), denominateur = 1 et signe = signe de n. Exemple : Fraction(-2) donne numerateur = 2, denominateur = 1 et signe = negatif.
- Un constructeur de copie classique.

En plus des setters/getters classiques, la classe Fraction doit disposer des 3 **méthodes d'instance** suivantes :

- **bool estPositif()** : qui retourne true si la fraction est strictement positive (numerateur ne peut pas être égal à 0), false sinon.
- **bool estNegatif()** : qui retourne true si la fraction est strictement négative (numerateur ne peut pas être égal à 0), false sinon.
- **bool estNul()** : qui retourne true si la fraction est nulle, c'est-à-dire si numerateur est égal à 0, false sinon.

De plus, la classe Fraction possèdera deux **variables membres statiques constantes** **UN** et **ZERO**, **instances de la classe Fraction**, représentant respectivement les nombres 1 (+1/1) et 0 (+0/1).

Dans le but futur de simplifier les fractions, nous avons besoin d'une méthode calculant le PGCD (Plus Grand Commun Diviseur) de deux nombres. On vous demande donc d'ajouter à la classe Fraction :

- La **méthode statique** **unsigned int PGCD(unsigned int a, unsigned int b)** permettant de calculer le PGCD de deux nombres entiers positifs a et b. Remarque : Voir algorithme d'Euclide pour le calcul du PGCD (Wikipédia est ton ami 😊).
- La **méthode d'instance** **void simplifie()** qui simplifie la fraction au maximum. Pour cela, il suffit de diviser le numérateur et le dénominateur de la fraction par le PGCD(numerateur, denominateur).

Nous pouvons à présent compléter la classe Fraction de plusieurs opérateurs.

c) Surcharge des opérateurs = et << de la classe Fraction (Essai5() et Essai6())

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

```
Fraction f, f1(…);
f = f1 ;
```

Ensuite, l'opérateur << devra respecter les contraintes suivantes :

- Si la fraction est nulle (numerateur == 0), on affichera la chaine de caractères « 0 », inutile d'afficher le dénominateur, ni le signe. Exemple : on affichera « 0 » et non « 0/7 » ou « -0/4 ».
- On affichera d'abord le signe. Si celui-ci est positif, on ne l'affichera pas. Exemples : « 3/5 », « -2/3 », …
- Si le dénominateur est égal à 1, on ne l'affichera pas. Exemples : On affichera « -3 » et non « -3/1 ». On affichera « 5 » et non « 5/1 », …

d) Surcharge de l'opérateur (Fraction + Fraction) de la classe Fraction (Essai6())

Pour rappel, pour additionner deux fractions, on doit tout d'abord les mettre sur le même dénominateur, puis faire l'addition :

$$\frac{a}{b}+\frac{c}{d}=\frac{ad}{bd}+\frac{bc}{bd}=\frac{ad+bc}{bd}$$

La fraction obtenue devra bien sûr être simplifiée au maximum ! On vous demande donc de surcharger l'opérateur + de la classe Fraction permettant d'exécuter un code du genre :

```
Fraction f1(3,10), f2(4,15), f, f3, f4 ;
f = f1 + f2 ;
f = f + Fraction(1,2,Fraction::negatif) ;
f3 = Fraction(-2) + f ;
```

e) Surcharge des opérateurs (Fraction + int) et (int + Fraction) de la classe Fraction (Essai6())

Pour rappel, pour additionner une fraction et un nombre entier, on doit tout d'abord mettre le nombre entier sur le même dénominateur que la fraction, puis faire l'addition :

$$\frac{a}{b}+c=\frac{a}{b}+\frac{bc}{b}=\frac{a+bc}{b}$$

La fraction obtenue devra bien sûr être simplifiée au maximum ! On vous demande donc de surcharger l'opérateur + de la classe Fraction permettant d'exécuter un code du genre :

```
Fraction f1(3,10), f, f2, f3 ;
f = f1 + 2 ;
f = Fraction(1,2,Fraction::negatif) + 5 ;
f3 = 3 + f ;
```

Suggestion : ré-utiliser l'opérateur + développé en d)

f) Surcharge de l'opérateur (Fraction - Fraction) de la classe Fraction (Essai7())

Pour rappel, pour soustraire deux fractions, on doit tout d'abord les mettre sur le même dénominateur, puis faire la soustraction :

$$\frac{a}{b}-\frac{c}{d}=\frac{ad}{bd}-\frac{bc}{bd}=\frac{ad-bc}{bd}$$

La fraction obtenue devra bien sûr être simplifiée au maximum ! On vous demande donc de surcharger l'opérateur - de la classe Fraction permettant d'exécuter un code du genre :

```
Fraction f1(3,10), f2(4,15), f, f3, f4 ;
f = f1 - f2 ;
f = f - Fraction(1,2,Fraction::negatif) ;
f3 = Fraction(-2) - f ;
```

g) Surcharge de l'opérateur (Fraction * Fraction) de la classe Fraction (Essai8())

Pour rappel, pour multiplier deux fractions, on doit multiplier leurs numérateurs et leurs dénominateurs entre eux :

$$\frac{a}{b} \times \frac{c}{d} = \frac{a \times c}{b \times d} = \frac{ac}{bd}$$

La fraction obtenue devra bien sûr être simplifiée au maximum ! On vous demande donc de surcharger l'opérateur * de la classe Fraction permettant d'exécuter un code du genre :

```
Fraction f1(3,10), f2(4,15), f, f3, f4 ;
f = f1 * f2 ;
f = f * Fraction(1,2,Fraction::negatif) ;
f3 = Fraction(-2) * f ;
```

h) Surcharge de l'opérateur (Fraction / Fraction) de la classe Fraction (Essai9())

Pour rappel, pour diviser deux fractions, on doit multiplier la première par l'inverse de la seconde :

$$\frac{a}{b} \div \frac{c}{d} = \frac{a}{b} \times \frac{d}{c} = \frac{ad}{bc}$$

La fraction obtenue devra bien sûr être simplifiée au maximum ! On vous demande donc de surcharger l'opérateur / de la classe Fraction permettant d'exécuter un code du genre :

```
Fraction f1(3,10), f2(4,15), f, f3, f4 ;
f = f1 / f2 ;
f = f / Fraction(1,2,Fraction::negatif) ;
f3 = Fraction(-2) / f ;
```

i) Surcharge des opérateurs de comparaison (< > et ==) de la classe Fraction (Essai10())

Pour rappel, deux fractions

$$\frac{a}{b} et \frac{c}{d} \ sont\ égales\ si\ a \times d = b \times c$$

Faites attention au signe ! On vous demande donc de surcharger l'opérateur == (mais également < et >) de la classe Fraction permettant d'exécuter un code du genre :

```
Fraction f1, f2;
…
if (f1 < f2) …
if (f1 > f2) …
if (f1 == f2) …
```

j) Surcharge des opérateurs ++ de la classe Fraction : Essai11()

On vous demande de programmer les opérateurs de post et pré-incrémentation de la classe Fraction. Ceux-ci incrémenteront une fraction de 1. Cela permettra d'exécuter le code suivant :

```
Fraction f(1,3,Fraction::negatif);

cout << ++f << endl ; // f vaut à présent 2/3
cout << f++ << endl ; // f vaut à présent 5/3
```

1.4 Jeu de tests 4 (Test4.cpp) :

Associations de classes : héritage et virtualité

On se rend vite compte qu'une interface graphique sans image et sans bouton pour interagir avec, ce n'est pas très joli ni très utile. Nous allons donc étendre les fonctionnalités de la classe Panneau afin de créer de nouvelles classes « graphiques ». Ainsi, on vous demande de développer la petite hiérarchie de classes décrite ci-dessous.

a) La classe PanneauImage (héritage simple) : Essai1()

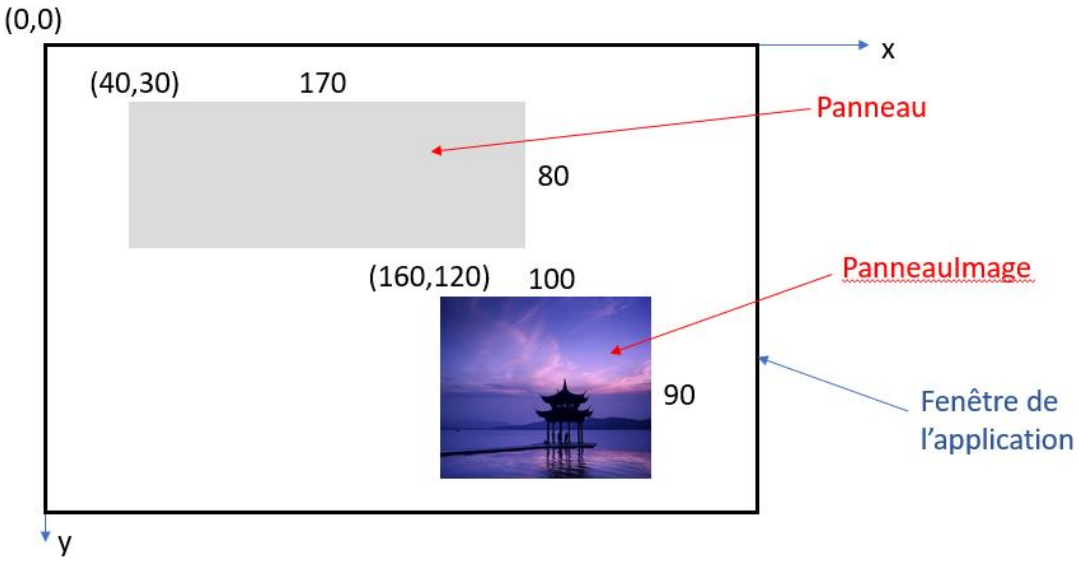
Un PanneauImage est un Panneau qui affichera une image à la place d'une zone de couleur uniforme. On vous demande donc de programmer la classe **PanneauImage**, héritant de la classe **Panneau** et qui :

- Possède ses propres constructeurs et opérateurs <<, >> et =
- Possède, en plus, une nouvelle variable membre de type chaine de caractères (**char***), **fichier**, et qui contiendra le nom du fichier image (une image bitmap dans la suite, voir seconde partie du dossier). La variable couleur n'étant plus utilisée, elle prendra une valeur par défaut. De plus, les variables largeur et hauteur prendront également (pour l'instant !) des valeurs par défaut (ces valeurs seront affectées aux dimensions de l'image lue dans le fichier bitmap → voir seconde partie du dossier à nouveau).
- Redéfinit la méthode **Affiche()** de telle sorte qu'elle affiche ce genre de chose :

```
[PANNEAU_IMAGE P01 : Position(10,40) fichier="image1.bmp"]
```

pour un PanneauImage dont le nom est P01, qui est situé aux coordonnées (10,40), et dont le nom de fichier associé est image1.bmp

Voici un exemple :



On observe donc :

- Un **Panneau** de position (x,y)=(40,30), de dimensions (largeur,hauteur)=(170,80) et de couleur grise.
- Un **PanneauImage** de position (x,y)=(160,120), de dimensions (largeur,hauteur)=(100,90) et d'image « paysage.bmp ». Comme déjà mentionné, dans la seconde partie de l'énoncé, vous recevrez des outils permettant d'ouvrir (à partir d'un nom de fichier), d'afficher une image bitmap et d'en récupérer les dimensions.

b) La classe Bouton (héritage simple) : Essai2()

Un Bouton est un Panneau sur lequel on pourra cliquer afin qu'une action s'exécute. On vous demande de programmer la classe **Bouton**, qui hérite de la classe **Panneau**, et qui présente en plus :

- Une méthode d'instance **void clic(int X,int Y)** qui (pour l'instant ! → voir seconde partie du dossier) affiche « Clic OK ! » lorsque le clic de coordonnées (X,Y) se trouve dans la zone délimitée par le bouton (c'est-à-dire si x <= X et X <= x+largeur et si y <= Y et Y <= y+hauteur), et « Clic à coté… » sinon. C'est cette méthode qui rend le bouton actif. Actuellement, aucun clic n'est possible car vous ne disposez pas encore de fenêtre graphique. Dans la seconde partie de l'énoncé, lorsque l'on cliquera dans la fenêtre graphique et que l'on récupèrera la position (X,Y) du clic, on pourra tester, grâce à la méthode clic(), si on a cliqué sur un bouton et donc exécuter une action particulière.
- Ses propres constructeurs et opérateurs <<, >> et =
- la méthode **Affiche()** qui affiche un truc du genre :

```
[BOUTON B01 : Position(30,90) Largeur=50 Hauteur=50 Couleur(0,255,0,vert)]
```

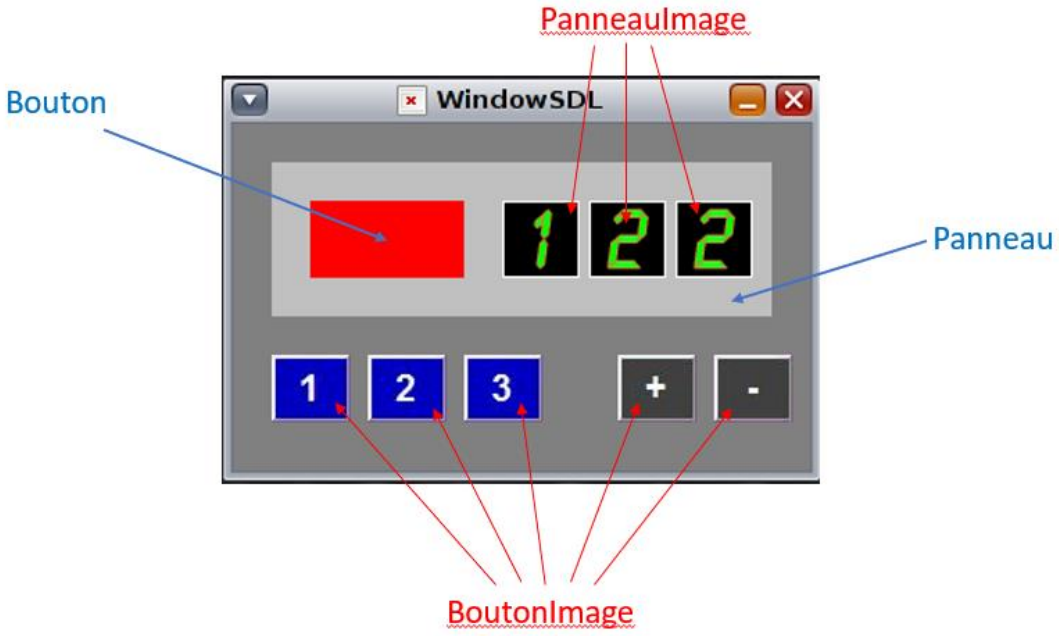
c) La classe BoutonImage (héritage multiple) : Essai3()

Il serait intéressant de disposer d'un Bouton qui affiche également une image à la place d'une zone de couleur uniforme. On vous demande donc de programmer la classe **BoutonImage**, qui hérite de la classe **Bouton** mais également de la classe **PanneauImage**, et qui présente en plus :

- Ses propres constructeurs et opérateurs <<, >> et =
- La méthode **Affiche()** qui affiche un truc du genre :

```
[BOUTON_IMAGE B02 : Position(40,300) fichier="boutonCancel.bmp"]
```

Voici un exemple simplifié d'interface graphique que vous pourrez construire dans la suite (2ème partie de l'énoncé) :



Dans la seconde partie de l'énoncé, vous ajouterez aux classes Panneau, PanneauImage, Bouton et BoutonImage les méthodes permettant des les afficher dans une fenêtre graphique. Les PanneauImages et les BoutonImages seront créés à partir de petits fichiers bitmap.

d) Mise en évidence de la virtualité et du down-casting : Essai4(), Essai5() et Essai6()

Dans un premier temps, on demande :

- de maintenir la méthode **Affiche()** **non-virtuelle** (et cela à des fins purement pédagogiques afin que vous compreniez la différence entre une méthode qui est virtuelle et une autre qui ne l'est pas).
- d'ajouter aux classes Panneau, PanneauImage, Bouton et BoutonImage la **méthode virtuelle getType()** qui retourne la chaîne de caractères « PANNEAU », « PANNEAU_IMAGE », « BOUTON » ou « BOUTON_IMAGE » selon le cas. Pour cela, on vous demande de mettre en place dans chacune de ces classes une **variable membre statique constante** (de type **char***) contenant le message qui sera renvoyé par la méthode getType().

Les méthodes **Affiche()** et **getType()** étant respectivement non-virtuelle et virtuelle, on vous demande de :

- comprendre et savoir expliquer le code et les résultats des essais 4 et 5 mettant en évidence la (non-)virtualité.
- comprendre et savoir expliquer le code et les résultats de l'essai 6 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 **chaine de caractères (char *)**. Celle-ci contiendra un message « utilisateur » lié à l'erreur. Elle est lancée lorsque le nom d'un panneau est invalide. Pour que le nom d'un **Panneau** 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 p est un objet de la classe Panneau, p.setNom("P") ou p.setNom("PX34") lancera une exception avec un message du genre « Nom invalide ! ». Un exemple de nom valide est "P17". En ce qui concerne le nom d'un **PanneauImage**, il en est de même. Pour **Bouton** et **BoutonImage**, idem sauf que la première lettre doit être un **'B'.** Vous devez donc **surcharger la méthode setNom()** dans les classes Panneau, PanneauImage, Bouton et BoutonImage tout en faisant attention à la **virtualité de la méthode… A vous de voir !!! Virtuelle ou pas ?**

- **FractionException** : lancée lorsque l'on tente de créer ou réaliser une opération invalide avec une Fraction. **Cette classe va hériter directement de BaseException** mais possèdera en plus :
  - Une variable membre **code** de type **int**, représentant un code d'erreur qui pourra être utile dans la suite pour le programmeur, et pouvant prendre les valeurs ERREUR_DENOMINATEUR_NUL ou ERREUR_DIVISION_PAR_ZERO qui sont deux variables membres statiques constantes de type int de la classe FractionException.
  - Les méthodes **getCode()** et **setCode()** associées à la variable code.
  - La méthode **Affiche()** permettant d'afficher le message utilisateur (chaine de caractères héritée de BaseException) et le code d'erreur.

  Par exemple, créer (ou saisir au clavier avec cin) une fraction ayant un dénominateur nul lancera l'exception FractionException dont le message utilisateur est « Dénominateur nul ! » par exemple et dont le code d'erreur est ERREUR_DENOMINATEUR_NUL. Diviser une fraction par une fraction dont le numérateur est nul (pour rappel, la classe Fraction possède une méthode **estNul()** qui pourrait être utile ici) lancera une exception dont le message utilisateur est « Division par 0 ! » par exemple et dont le code d'erreur est ERREUR_DIVISION_PAR_ZERO.

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 tous les panneaux et boutons de l'interface graphique ou encore toutes les fractions manipulées par notre calculatrice. Nous allons ici mettre en place une base pour nos containers. Ceux-ci seront construits via une hiérarchie de classes templates.

b) Le container typique : la liste

Le cœur de notre hiérarchie va être une liste chaînée dynamique. Pour rappel, une liste chaînée dynamique présente un pointeur de tête et une succession de cellules liées entre elles par des pointeurs, la dernière cellule pointant vers NULL. Cette liste va être encapsulée dans une **classe abstraite ListeBase template** contenant comme seule variable membre le pointeur de tête de la liste chaînée. Elle aura donc la structure de base suivante :

```
template<class T> class ListeBase
{
	Protected :
		Cellule<T> *pTete ;
	…
}
```

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

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

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

- Un **constructeur par défaut** permettant d'initialiser le pointeur de tête à NULL.
- Un **constructeur de copie**.
- Un **destructeur** permettant de libérer correctement la mémoire.
- La méthode **estVide()** retournant le booléen true si la liste est vide et false sinon.
- La méthode **getNombreElements()** retournant le nombre d'éléments présents dans la liste.
- La méthode **Affiche()** permettant de parcourir la liste et d'afficher chaque élément de celle-ci.

- La **méthode virtuelle pure** **void insere(const T & val)** qui permettra, une fois redéfinie dans une classe héritée, d'insérer un nouvel élément dans la liste, à un endroit dépendant du genre de liste héritée (simple liste, pile, file, liste triée, …).
- Un **opérateur =** permettant de réaliser l'opération « liste1 = liste2 ; » sans altérer la liste2 et de telle sorte que si la liste1 est modifiée, la liste2 ne l'est pas et réciproquement.
- Un **opérateur [ ]** permettant de chercher et de retourner un élément particulier de la liste. Par exemple, « liste[3] » devra retourner l'élément d'indice 3 de la liste (les indices commençant à 0) sans altérer la liste.

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 **PanneauImage**.

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 ListeTriee** 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 ListeTriee avec des **entiers**, puis ensuite avec des objets de la classe **Fraction**. Celles-ci devront bien sûr être triés par ordre croissant.

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 de les afficher ou de faire une recherche. La notion d'itérateur va nous permettre de réaliser ces opérations.

On vous demande donc de créer la classe **Iterateur** qui sera un **itérateur** de la classe **ListeBase** (elle permettra donc de parcourir tout objet instanciant la classe Liste ou 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.

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

f) BONUS (pour la 1ère partie) : La pile

Dans la seconde partie de l'énoncé, vous aurez besoin d'une pile. Donc, si vous avez le temps dès maintenant (non obligatoire par la 1ère évaluation donc !), 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 chainée, 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 **Fraction**. Sans aucune modification, l'itérateur développé précédemment devra également être utilisable avec la classe Pile.

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 PanneauImage se sérialise elle-même

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

- **Save(ofstream & fichier) const** permettant d'enregistrer sur flux fichier toutes les données d'un PanneauImage (nom, position, dimensions et nom de fichier image) 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 un PanneauImage enregistré 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 **Panneau**, et **Couleur** les méthodes suivantes :

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

Ces méthodes seront appelées par les méthodes Save et Load de la classe PanneauImage/Panneau lorsqu'elle devra enregistrer ou lire ses variables membres dont le type n'est pas un type de base.

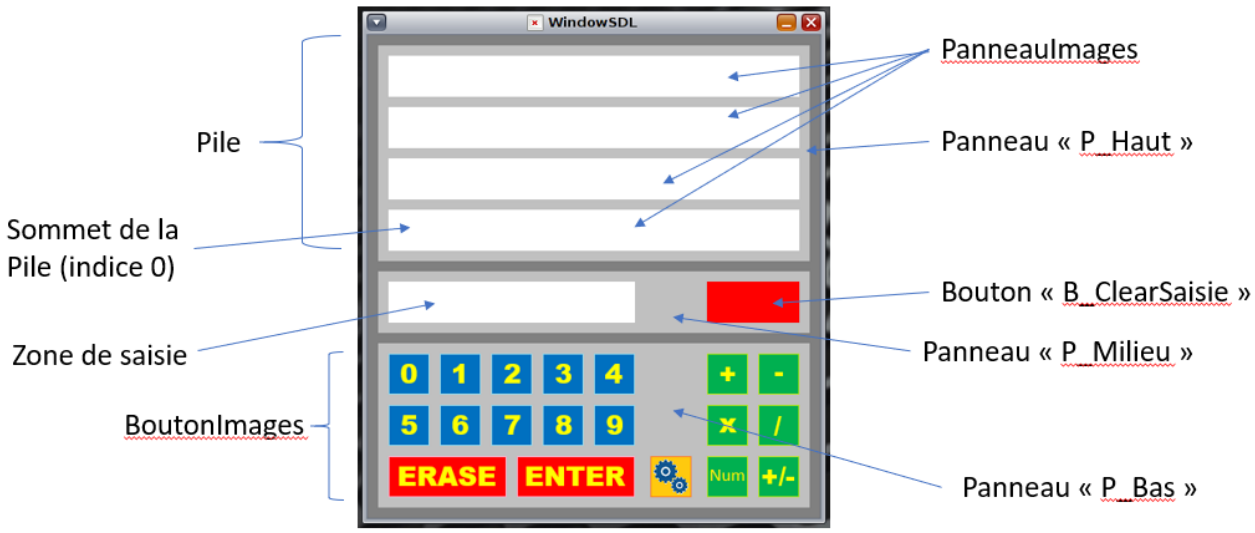
Tous les enregistrements seront de taille variable. Pour l'enregistrement d'une chaine de caractères « chaine » (type **char ***), on enregistrera tout d'abord le nombre de caractères de la chaine (strlen(chaine)) puis ensuite la chaine elle-même. Ainsi, lors de la lecture dans le fichier, on lit tout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

2. Deuxième Partie : Développement de l'application

Points principaux de l'évaluation	subdivisions
EVALUATION 2 (Développement de l'application) → Examen de Janvier 2019	
Classe Calculette	Constructeurs
	Pile<Fraction>
	Méthodes Plus(), …, getPile(int i)
	Lancement de l'exception **CalculetteException**
	TestCalculette.cpp OK ?
Classe Fenetre	Utilisation de la librairie WindowSDL
	Classe **Fenetre**. Méthode virtuelle **Dessine()**
	Mise à jour des classes **Panneau**, **PanneauImage**, **Bouton** et **BoutonImage** : méthode **Dessine()**
	TestFenetre.cpp OK ?
Classe FenetreCalculette	**Liste<Panneau>, Liste<BoutonImage>, PanneauImage[][]**
	Constructeur par défaut
	Méthode **Dessine()**
	Méthodes set/getCouleurPanneauXXX
	Utilisation itérateur ?
	TestFenetreCalculette.cpp OK ?
Classe Controleur	Classe abstraite **Ecouteur**
	Modification de la classe **Bouton**
	Agrégation par référence Controleur – Calculette - FenetreCalculette
	Redéfinition de la méthode **ActionBouton**
	Mise à jour de **FenetreApplication** : **Dessine()** et **clic()**
	Gestion de l'exception **CalculetteException**
Sérialisation	Gestion du fichier texte **« boutons.csv »**
	Mise à jour du constructeur de **FenetreCalculette**
	Gestion du fichier binaire **« config.dat »**
Menu de configuration	Mise à jour de **« config.dat »** ?
BONUS (non obligatoire)	Autres boutons ?
	Autres fonctionnalités dans le menu
	Une fenêtre spécifique en cas d'exception ?
	FenetreCalculette est son propre Ecouteur ?

2.1 L'application proprement dite : Premier aperçu

Maintenant que nous disposons des classes de base essentielles à notre application, nous pouvons nous attaquer à l'application proprement dite. Au démarrage, la fenêtre graphique suivante devra apparaître :



La calculette dispose de 3 panneaux :

- Le panneau du haut dont le nom est « P_Haut » et qui contient 4 lignes d'affichage. Chaque ligne d'affichage est constituée de 10 PanneauImage pouvant contenir l'image d'un chiffre (ou un vide). Cette zone permettra d'afficher le contenu de la mémoire de la calculatrice. Celle-ci est en fait une pile de fractions dont le sommet (indice 0) est la ligne d'affichage du bas.
- Le panneau du milieu dont le nom est « P_Milieu » et qui contient une zone de saisie et un bouton rouge. La zone de saisie sert de zone tampon permettant à l'utilisateur d'encoder un nombre avant de le transférer vers la pile. Cette zone de saisie est constituée de 6 PanneauImage pouvant contenir l'image d'un chiffre (ou un vide). Le bouton rouge dont le nom est « B_ClearSaisie » permet de vider la zone de saisie (sans transfert vers la pile).
- Le panneau du bas dont le nom est « P_Bas » et qui contient l'ensemble des boutons permettant d'interagir avec la calculette. Les 10 boutons bleus permettent de saisir un nombre dans la zone de saisie. Le bouton « ENTER » permet de transférer le nombre de la zone de saisie au sommet de la pile (à l'indice 0). Le bouton « ERASE » vide la mémoire de la calculette (pile + zone de saisie). Les boutons verts permettent de réaliser des opérations sur les fractions stockées dans la pile. Les opérations + - * / dépilent les 2 fractions au sommet de la pile, réalisent leur opération (pile[1] op pile[0]) et empilent le résultat. Le bouton « +/- » change le signe (calcule l'opposé) de la fraction située au sommet de la pile. Le bouton « Num » permet d'afficher toutes les fractions de la pile sous forme numérique (1/3 ↔ 0.33333).

Comme déjà dit, la calculette fonctionne selon la notation polonaise inversée. Le plus simple est de l'illustrer sur un exemple (voir ci-dessous).

Remarque : Vous remarquerez que le nom des boutons et des panneaux ne respecte pas les contraintes (voir jeu de tests 5). Cependant, afin d'avoir des noms plus explicites (que de simples numéros), on vous demande de modifier votre code en conséquence (« P_... » ou « B_... »).

Afin d'illustrer le fonctionnement, supposons que nous voulions réaliser le calcul suivant :

$$\frac{3}{10} + \frac{2}{5} = ?$$

La séquence des touches à enfoncer est la suivante :

Le développement de la calculette va se faire en plusieurs étapes :

1. Construction de la logique métier (classe **Calculette** gérant toute la logique et la mémoire de la calculatrice sans tenir compte de l'interface graphique).
2. Construction de l'interface graphique (classes **Fenetre** et **FenetreCalculette**).
3. Mise en relation entre la classe Calculette et la classe FenetreCalculette (classe **Controleur**).

2.2 Construction de la logique « métier » : la classe Calculette

La classe calculette va contenir la mémoire de la calculatrice, c'est-à-dire une pile de fractions mais également la zone de saisie constituée d'une chaîne de caractères. De plus, elle va contenir, entre autres, toutes les méthodes qui seront appelées lorsque l'on cliquera sur un des boutons.

Votre classe **Calculette** doit comporter les variables membres suivantes :

- **Pile<Fraction> pile** : qui représente la mémoire de la calculatrice.
- **char saisie[10]** : qui représente la zone de saisie sous forme d'une chaine de caractères.
- **bool mode** : un booléen valant true si les fractions doivent être affichées sous forme fractionnaire (ex : 1/3) et false si elles doivent être affichées sous forme numérique (ex : 0.33333).

En plus d'un constructeur par défaut et de copie, la classe **Calculette** disposera des méthodes suivantes :

- **Affiche()** : permettant d'afficher dans la console le contenu de la pile et de la saisie.
- **SaisirChiffre(unsigned int val)** : permettant de saisir un chiffre, c'est-à-dire de le concaténer à la chaine de caractères saisie actuelle.
- **ClearSaisie()** : permettant de vider la chaine de caractères saisie.
- **Plus(), Moins(), Fois(), divise()** permettant de réaliser les opérations + - * / sur les deux fractions situées au sommet de la pile. Attention à l'ordre des opérandes :

```
Fraction b = pile.pop() ;
Fraction a = pile.pop() ;
Fraction res = a / b ;    // pour la méthode Divise()
pile.push(res) ;
```

- **PlusOuMoins()** : permettant de changer le signe de la fraction située au sommet de la pile.
- **Erase()** : permettant de vider la pile et la chaine de caractères saisie.
- **ChangeMode()** : permettant de changer le mode (true ↔ false).
- **getSaisie()** : permettant de retourner la **chaine de caractères** saisie (celle-ci pourra être affichée plus tard dans l'interface graphique).
- **getPile(int i)** : permettant de retourner une **chaine de caractères** représentant la fraction située à l'indice i de la pile (l'indice 0 correspondant au sommet de la pile). Si mode = true, la chaine de caractères retournée contiendra la fraction sous la même forme que celle fournie par l'opérateur << de la classe Fraction. Si mode = false, la chaine de caractères retournée contiendra le résultat de la division du numérateur par le dénominateur de la fraction.

Si la pile ne contient pas un nombre suffisant de fractions (2 pour + - * /, 1 pour « +/- »), les méthodes Plus(), Moins(), Fois(), Divise(), PlusOuMoins() devront lancer l'exception **CalculetteException** qui hérite de **BaseException**. De même, Divise() lancera l'exception **CalculetteException** si on tente de diviser une fraction par zéro.

Afin de vous aider à mettre au point vos classes Calculette et CalculetteException, on vous fournit le programme **TestCalculette.cpp**.

2.3 Finalisation de notre librairie graphique : la classe Fenetre

Il est maintenant temps de doter nos classes Panneau, PanneauImage, Bouton et BoutonImage des capacités d'apparaître dans une fenêtre graphique.

Pour cela, on vous donne une petite librairie fournie sous la forme de 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. Ce sera la méthode **FillRectangle()** qui nous intéressera pour l'affichage de nos panneaux.
- **void drawImage(WindowSDLimage image, int x,int y)** qui permet d'afficher une image à l'emplacement (x,y) de la fenêtre graphique. La classe **WindowSDLimage** gère pour vous le chargement d'un fichier bitmap en mémoire.
- 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)**.

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.

On vous demande à présent de créer la classe **Fenetre** qui représente une simple fenêtre vide dont on peut juste spécifier les dimensions et la couleur du fond. Elle dispose donc des variables membres suivantes :

- **int largeur, hauteur** : qui définissent les dimensions de la fenêtre en pixels
- **Couleur couleurFond** : qui définit la couleur de fond de la fenêtre

Elle dispose des méthodes suivantes :

- Les trois types de constructeurs classiques (par défaut, initialisation, copie). Attention, ces constructeurs ne font qu'initialiser les variables membres, ils n'ouvrent pas la fenêtre graphique !
- Des accesseurs get/setXXX traditionnels associées aux variables membres.
- La méthode **setVisible(bool v)** : qui ouvre la fenêtre si v = true, et qui ferme la fenêtre si v = false. C'est donc cette méthode qui est responsable de l'apparition/disparition de la fenêtre graphique. Pour cela, elle utilise les méthodes open() et close() de la classe WindowSDL.
- La méthode **virtuelle** **Dessine()** : qui redessine le contenu de la fenêtre. Ici, il s'agit simplement d'afficher la couleur de fond de la fenêtre (utilisation de la méthode setBackground() de WindowSDL).

Nous allons à présent mettre à jour notre classe **Panneau** afin qu'elle puisse être puisse être affichée dans la fenêtre graphique. Pour cela, on vous demande de lui ajouter la méthode

- **Dessine()** : qui permet à un Panneau d'être dessiné dans la fenêtre graphique. Pour cela, vous devez utiliser la méthode fillRectangle() de la classe WindowSDL.

Venons-en à la classe **PanneauImage**. Vous devez lui ajouter à la variable membre

- **WindowSDLimage * image** : il s'agit d'un pointeur vers un objet de type WindowSDLimage qui sera alloué dynamiquement.

Vous devez également mettre à jour les constructeurs (si nécessaire) et surtout la méthode **setFichier(const char* fichier)** de la classe **PanneauImage**. Sur base du nom de fichier, un objet WindowSDLimage doit être instancié (la classe WindowSDLimage contient un constructeur qui fait le travail pour vous). Si le fichier n'existe pas, le pointeur **image** doit être mis à NULL. Une fois l'image chargée en mémoire, les variables largeur et hauteur de la classe **PanneauImage** doivent être mises à jour en utilisant les méthodes getWidth() et getHeight() de la classe WindowSDLimage.

Enfin, vous devez rajouter à la classe **PanneauImage** la méthode

- **Dessine()** : qui permet à un PanneauImage d'être dessiné dans la fenêtre graphique. Pour cela, vous devez utiliser la méthode drawImage() de la classe WindowSDL. Dans le cas où la variable image est NULL, vous devez afficher un rectangle noir en remplacement.

Pour la classe **Bouton**, rien à faire (normalement…) vu qu'elle hérite de Panneau.

Pour la classe **BoutonImage**, rien à faire (normalement…) vu qu'elle hérite de PanneauImage.

Afin de mettre au point vos classes, on vous donne le programme **TestFenetre.cpp**.

Remarque : Pour l'instant, les boutons ne réalisent aucune action lorsque l'on clique dessus. La méthode clic() de la classe Bouton sera mise à jour plus tard (voir ci-dessous).

2.4 Construction de l'interface graphique : la classe FenetreCalculette

Nous allons à présent créer l'interface graphique de notre calculatrice. Pour cela, nous allons partir de la classe **Fenetre** (qui est une fenêtre graphique vide) et la spécialiser en lui donnant une taille spécifique et en lui ajoutant les composants graphiques utiles comme les boutons et les panneaux.

On vous demande de créer la classe **FenetreCalculette**, qui hérite de **Fenetre** et dont la structure générale est la suivante :

```
class FenetreCalculette : public Fenetre
{
  private :
    Liste<Panneau>      listePanneaux ;
    Liste<BoutonImage>  listeBoutons ;
    Bouton              boutonClearSaisie ;
    PanneauImage        affichage[4][10] ;
    PanneauImage        saisie[6] ;
  public :
    …
} ;
```

Cette classe dispose donc des variables membres suivantes :

- **listePanneaux** : qui contient l'ensemble des panneaux de l'application.
- **listeBoutons** : qui contient l'ensemble des BoutonImages de l'application.
- **boutonClearSaisie** : il s'agit du bouton rouge permettant de vider la zone de saisie.
- **affichage** : qui correspond aux 4 lignes d'affichage du panneau du haut. Chaque ligne dispose de 10 PanneauImages.
- **saisie** : qui correspond à la zone de saisie située dans le panneau du milieu. Elle est constituée de 6 PanneauImages.

La classe **FenetreCalculette** doit disposer d'un constructeur par défaut qui

- fixe la **largeur** de la fenêtre à 440 et sa **hauteur** à 470,
- affecte la valeur de la **couleur de fond** à un gris de valeur 128,
- ajoute à **listePanneaux** les trois panneaux dont on a déjà parlé. En voici les caractéristiques :

Nom	x	y	largeur	hauteur	Couleur
"P_Haut"	10	10	420	210	Couleur::ARGENT
"P_Milieu"	10	230	420	60	Couleur::ARGENT
"P_Bas"	10	300	420	160	Couleur::ARGENT

- initialise **boutonClearSaisie** avec les champs nom= « B_ClearSaisie », x=330, y=240, largeur=90, hauteur=40 et couleur=Couleur(255,0,0).

- initialise les 40 PanneauImages de la variable **affichage** avec un fichier image dont le nom est « images/Vide.bmp ». Il s'agit d'une petite image blanche de taille 40 pixels × 40 pixels. Etant tous alignés côte à côte, les positions horizontales (x) des panneaux sont 20, 60, 100, 140, 180, 220, 260, 300, 340, 380. Etant alignés sur 4 lignes, les positions verticales (y) des panneaux sont 170, 120, 70, 20.
- initialise les 6 PanneauImages de la variable **saisie** avec le fichier dont le nom est « images/Vide.bmp ». Ces 6 panneaux sont tous à la même hauteur et ont une position verticale (y) égale à 240. Etant tous alignés côte à côte, les positions horizontales (x) des 6 panneaux sont 20, 60, 100, 140, 180, 220.
- ajoute à **listeBoutons** l'ensemble des boutons du panneau du bas. En voici les caractéristiques :

Nom	x	y	fichier
"B_0"	20	310	"images/boutonZero.bmp"
"B_1"	70	310	"images/boutonUn.bmp"
"B_2"	120	310	"images/boutonDeux.bmp"
"B_3"	170	310	"images/boutonTrois.bmp"
"B_4"	220	310	"images/boutonQuatre.bmp"
"B_5"	20	360	"images/boutonCinq.bmp"
"B_6"	70	360	"images/boutonSix.bmp"
"B_7"	120	360	"images/boutonSept.bmp"
"B_8"	170	360	"images/boutonHuit.bmp"
"B_9"	220	360	"images/boutonNeuf.bmp"
"B_Erase"	20	410	"images/boutonErase.bmp"
"B_Enter"	145	410	"images/boutonEnter.bmp"
"B_Plus"	330	310	"images/boutonPlus.bmp"
"B_Moins"	380	310	"images/boutonMoins.bmp"
"B_Fois"	330	360	"images/boutonFois.bmp"
"B_Divise"	380	360	"images/boutonDivise.bmp"
"B_PlusOuMoins"	380	410	"images/boutonPlusOuMoins.bmp"
"B_Numerique"	330	410	"images/boutonNumerique.bmp"
"B_Config"	275	410	"images/boutonConfig.bmp"

Vous devez ensuite redéfinir la méthode **Dessine()** de la classe **FenetreCalculette** dans laquelle :

- La méthode **Dessine()** de la classe **Fenetre** est appelée, cela afin de redessiner le fond.
- Vous devez appeler la méthode **Dessine()** de chaque composant de la fenêtre. Pour cela, vous devez donc parcourir les conteneurs de boutons et de panneaux à l'aide d'itérateurs.

Enfin, on vous demande d'ajouter à la classe **FenetreCalculette** les méthodes :

- setCouleurPanneauHaut(const Couleur &), setCouleurPanneauMilieu(const Couleur &) et setCouleurPanneauBas(const Couleur &) qui modifient la couleur du panneau concerné.
- getCouleurPanneauHaut(), getCouleurPanneauMilieu() et getCouleurPanneauBas() qui retournent la couleur du panneau concerné.

Afin de mettre au point cette classe, vous disposez du programme **TestFenetreCalculette.cpp**.

2.5 Lien entre Calculette et FenetreCalculette : la classe Controleur

Nous disposons à présent des deux éléments essentiels de notre application :

- La classe **Calculette** qui contient toute la logique « métier » de la calculatrice.
- La classe **FenetreCalculatrice** qui représente l'interface graphique mais qui n'est pour l'instant qu'une « coquille vide » car les boutons ne réagissent pas et les zones d'affichage sont vides.

Le but à présent est de relier les deux afin d'obtenir notre application finale. Ce lien va être modélisé par la classe **Controleur** (voir plus bas) dont le rôle est de manipuler une instance de ces deux classes mais également de réagir aux clics sur les boutons.

Associer une action à un bouton : la classe Ecouteur

Le principe est de séparer la classe Bouton de l'action qui est réalisée lorsque l'on clique dessus. La méthode clic() de la classe Bouton va donc simplement exécuter une méthode précise d'un objet dont le rôle est justement de contenir l'action à exécuter. Cet objet sera une instance de la classe **Ecouteur** dont voici la structure

```
class Ecouteur
{
  public :
    virtual void ActionBouton(const char* nom) = 0 ;
} ;
```

Il s'agit d'une classe abstraite dont il faudra créer une classe dérivée dans la suite. Toute classe qui va hériter de la classe Ecouteur pourra se mettre « à l'écoute » d'un bouton, exécuter la méthode ActionBouton, c'est-à-dire exécuter la bonne action en fonction du nom du bouton reçu en paramètre. Pour cela, on vous demande de modifier la classe **Bouton** de la manière suivante :

- ajouter une variable membre **ecouteur** de type **Ecouteur*** : ce pointeur pointera vers un objet, héritant de la classe Ecouteur, et dont le rôle sera « d'écouter » les « événements » (dans notre cas, les « clics ») du bouton.
- modifier la méthode **clic(int X,int Y)** de telle sorte que, lorsqu'il y a « clic », c'est-à-dire lorsque (X,Y) se trouve à l'intérieur de la zone du bouton, la méthode ActionBouton() de **ecouteur** soit exécutée en lui passant en paramètre le nom du bouton.
- ajouter les méthodes **setEcouteur()** et **getEcouteur()** correspondant à la variable **ecouteur**.

L'avantage de cette manière de faire est que lorsque vous voulez associer une action à un bouton, vous ne devez pas modifier le code la classe Bouton ! Il vous suffit de créer une classe (héritant de la classe Ecouteur) qui contient l'action et de l'affecter (par référence → ecouteur est un pointeur vers un objet « extérieur ») au bouton.

Un seul Ecouteur pour tous nos boutons : la classe Controleur

Pour notre application, nous allons créer une seule classe qui sera à l'écoute de l'ensemble des boutons de la calculatrice, il s'agit de la classe **Controleur**. Etant donné qu'elle devra manipuler un objet calculette, elle devra contenir une référence (logique → un pointeur ici) vers un tel objet. De plus, cette même classe **Controleur** devra également être capable de contrôler l'interface graphique (pour lui demander de se redessiner par exemple), elle devra donc également contenir une référence (logique → un pointeur ici) vers un objet de classe FenetreCalculette.

On vous demande donc de créer la classe **Controleur**, qui hérite de la classe **Ecouteur**, et qui a la structure suivante

```
class Controleur : public Ecouteur
{
  private :
    Calculette * calculette ;
    FenetreCalculette * fenetre ;
  public :
    void ActionBouton(const char* nom) ;
    ….
} ;
```

Nous avons ici affaire à de l'agrégation par référence : Un objet de la classe Controleur ne contient pas d'objet Calculette ou FenetreCalculette, mais simplement des pointeurs vers des instances de ces classes instanciées autre part. Le destructeur de la classe **Controleur** ne devra donc pas faire de delete sur ces pointeurs !!!

La classe **Controleur** fait en quelque sorte office de « chef d'orchestre », ou de « contrôleur » de toute l'application. Elle réagit aux clics des boutons en manipulant soit l'objet pointé par calculette, soit celui pointé par fenetre.

Dans la classe **Controleur**, vous devez

- ajouter un constructeur par défaut mettant à NULL les deux pointeurs.
- ajouter un constructeur d'initialisation.
- ajouter les get/setXXX associées aux deux variables membres.
- redéfinir la méthode **ActionBouton** de telle sorte qu'elle fasse le lien entre le bouton (son nom plus exactement) et l'action à réaliser sur la calculatrice. Par exemple, si le nom du bouton est « B_3 », vous devez exécuter calculette->SaisirChiffre(3) ; si le nom du bouton est « B_Plus », vous devez exécuter calculette->Plus() ; si le nom du bouton est « B_Erase », vous devez exécuter calculette->Erase(), etc… Une fois l'action réalisée sur l'objet Calculette, la méthode **ActionBouton** doit demander à la fenêtre de se redessiner (**fenetre->Dessine()**).

Affichage des résultats des actions dans l'interface graphique

Actuellement, la méthode **Dessine()** de la classe **FenetreCalculette** ne fait qu'afficher tous les composants graphiques sans tenir compte des données contenues dans la calculatrice. Il nous faut donc mettre à jour la méthode **Dessine()** de la classe **FenetreCalculette**. Pour cela, on vous demande de modifier la classe **FenetreCalculatrice** de la manière suivante :

- lui ajouter une variable membre **calculette** de type **Calculette*** : il s'agit d'un pointeur vers l'objet de la classe Calculette. Via ce pointeur, la classe FenetreCalculette va pouvoir accéder à la mémoire de la calculatrice.
- lui ajouter les méthodes **getCalculette()** et **setCalculette()** associées à cette variable membre.
- mettre à jour la méthode **Dessine()** qui, après avoir dessiné tous les composants (ce qu'elle faisait déjà), met à jour les PanneauxImages des variables affichage[4][10] et saisie[6]. Pour cela, elle dispose des méthodes **getSaisie()** et **getPile(i)** de la classe Calculette. En outre, vous disposez des images « images/Un.bmp », « images/Deux.bmp », etc pour mettre à jour les PanneauImages.
- lui ajouter une méthode **void clic(int X,int Y)** qui ne fait qu'exécuter la méthode clic() de tous les boutons présents dans la fenêtre.

Mise en route de toute la machinerie

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

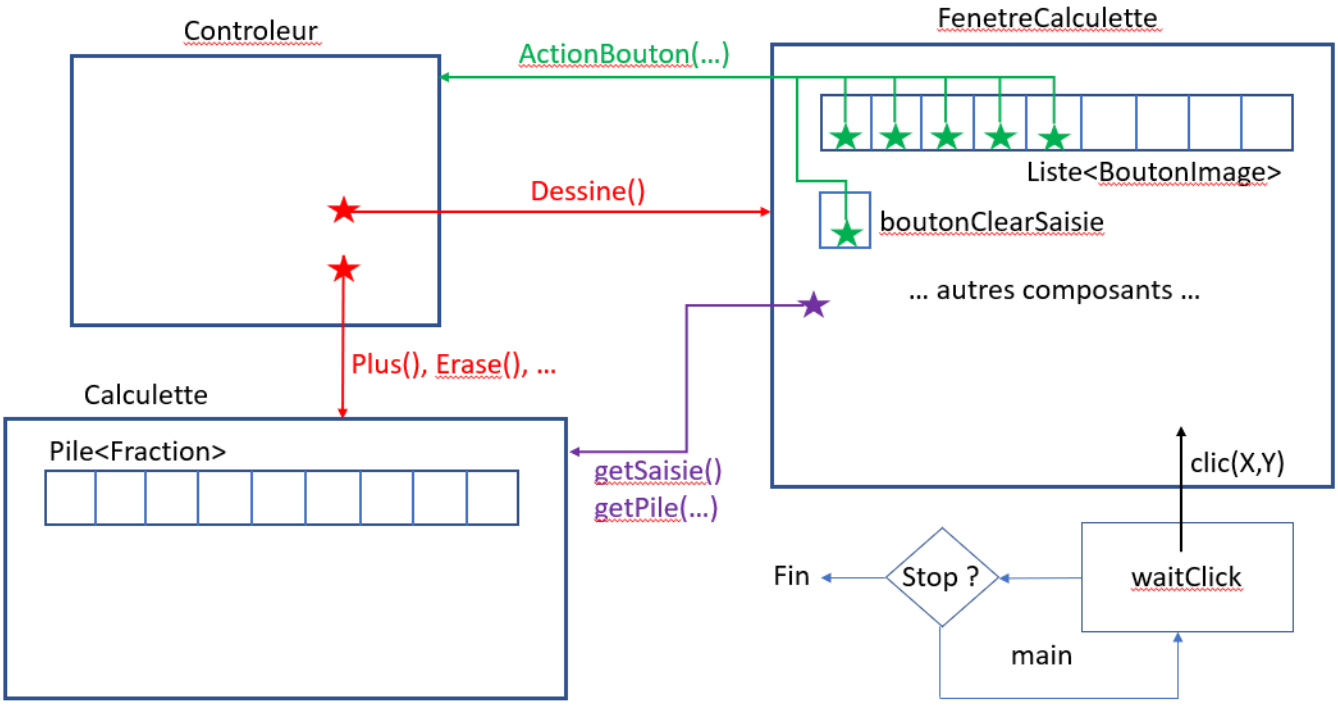
```
int main()
{
  Calculette calculette ;
  FenetreCalculette fenetre ;

  Controleur controleur(&calculette,&fenetre) ;

  fenetre.setEcouteur(controleur);       // qui relie chaque bouton au contrôleur
  fenetre.setCalculette(&calculette) ;   // qui relie la fenêtre à la calculette

  while(…)  // on appelle cette boucle la boucle des « événements »
  {
    WindowSDLclick clic = WindowSDL::waitClick() ;
    fenetre.clic(clic.getX(),clic.getY());
  }
}
```

Voici schématiquement la relation entre les différents objets et leurs interactions :



Remarque : En architecture logicielle, on parle de modèle « Model View Controler » (MVC) :

- Le « Model » représente les données ou encore la logique métier de l'application. Ici il s'agit de la classe Calculette.
- La « View » représente la vue ou encore l'interface (graphique généralement) entre l'utilisateur et le modèle. Il s'agit ici de la classe FenetreCalculette.
- Le « Controler » représente l'élément qui coordonne l'ensemble, qui fait donc le lien entre le modèle et la vue. Il s'agit ici de la classe Controleur.

Et si l'exception CalculetteException est lancée ?

A ce stade, la calculatrice doit fonctionner sans problème. Cependant, lorsque l'on tente de diviser par zéro ou que l'on tente de faire une opération alors que la pile ne contient pas assez d'éléments, l'application se termine car une exception CalculetteException est lancée et non traitée.

On vous demande donc de traiter correctement cette exception (dans la classe Controleur) de telle sorte que, lorsqu'elle est lancée,

- les trois panneaux de l'interface graphique apparaissent en rouge pendant 1 seconde (utilisation de **sleep()**). Pour cela, vous disposez des méthodes setCouleurPanneauXXX de la classe FenetreCalculette.
- La pile doit rester intacte, c'est-à-dire si des éléments sont dépilés mais que l'exception est lancée (suite à une division par zéro par exemple), les éléments doivent être replacés sur la pile.

2.6 Un fichier csv pour les boutons et sauvegarde de l'état de la calculette dans un fichier binaire

Personnalisation des boutons du panneau du bas : un fichier texte (csv)

Actuellement, les boutons du panneau du bas sont paramétrés (position et fichier de l'image) « en dur » dans l'application. Il serait intéressant de pouvoir personnaliser ces boutons en pouvant choisir leur position et l'image associée, et ce sans devoir tout recompiler à chaque fois. Pour cela, l'application, au démarrage, va lire un fichier texte dont le contenu est

```
B_0;20;310;images/boutonZero.bmp
B_1;70;310;images/boutonUn.bmp
…
B_Erase;20;410;images/boutonErase.bmp
…
B_Config;275;410;images/boutonConfig.bmp
```

Chaque ligne de ce fichier fournit les caractéristiques d'un des boutonImages du panneau du bas de la calculatrice (ces caractéristiques correspondent à celles fournies à la section 2.4). Il s'agit en fait d'un fichier csv dont le caractère **';'** est appelé le ***séparateur***. Celui-ci pourrait être **':'** ou encore **','**. Ce type de fichier peut être ouvert dans n'importe quel éditeur de texte ou Excel afin de pouvoir le modifier.

Dès lors, au démarrage, l'application doit

- tenter d'ouvrir en lecture le fichier **« boutons.csv »**. Si celui-ci n'existe pas, il doit être créé par l'application et « bidonné » avec les boutons « par défaut » dont les caractéristiques sont données à la section 2.4, puis ensuite être ouvert en lecture.
- lire le fichier ligne par ligne afin de créer les BoutonImages de l'application et de les insérer dans le conteneur **Liste<BoutonImage>** de la classe **FenetreCalculette**. Cela doit se faire dans le constructeur de la classe FenetreCalculette.

Tous les boutons du panneau du bas de la calculette sont à présent paramétrables par l'utilisateur. Il lui suffit pour cela de modifier le fichier **« boutons.csv »** afin de placer les boutons à sa guise et de leur attribuer l'image de son choix. **Attention !** Notez que le nom des boutons ne peut pas être modifié.

Le répertoire **images2** qui vous est fourni contient des autres images de boutons ainsi qu'un fichier csv permettant de tester une autre configuration que celle par défaut.

Sauvegarde de l'état de la calculatrice : un fichier binaire

Lorsque l'on quitte l'application en cliquant sur la croix de la fenêtre graphique, l'application doit sauvegarder l'état de la calculatrice dans un fichier binaire qui sera, par la suite, chargé au démarrage.

On vous demande donc de

- ajouter les méthodes **Save(ofstream &)** et **Load (ifstream)** à la classe **Fraction** afin qu'elle se (de)sérialise elle-même (au format binaire).
- ajouter les méthodes **Save(ofstream &)** et **Load(ifstream &)** à la classe **Calculette** afin qu'elle (de)sérialise son état (au format binaire), c'est-à-dire le **contenu de la pile**, le **contenu de la saisie** et le **mode**.
- Faire en sorte que lorsque l'on clique sur la croix, l'application enregistre l'état de la calculette dans le fichier binaire **« config.dat »**.
- Au démarrage, l'application doit tenter d'ouvrir le fichier **« config.dat »** afin de restaurer l'état de la calculette. Si celui-ci n'existe pas, une calculette « vierge » doit être utilisée.

2.7 Un petit menu de configuration des couleurs de la calculatrice

Lorsque l'on clique sur le bouton « B_Config » (le bouton orange), un petit menu doit apparaître dans le terminal :

```
***** MENU CONFIGURATION ************************************
  1) Changer la couleur du fond
  2) Changer la couleur du panneau du haut
  3) Changer la couleur du panneau du milieu
  4) Changer la couleur du panneau du bas
  q) Retour à la calculette
************************************************************
Choix :
```

Vous devez donc modifier la classe **Controleur** en conséquence. Pendant que l'utilisateur utilise le menu, il lui est impossible d'utiliser la calculette car l'application tourne dans la boucle du menu et non dans la boucle des « événements ».

Les 4 items du menu sont assez évidents et ne nécessitent pas d'explication. Notez que lorsque l'application se termine (clic sur la croix de la fenêtre), les 4 couleurs doivent également être enregistrées dans le fichier **« config.dat »** et donc rechargées au démarrage. Ces 4 couleurs font donc partie de l'état de la calculette.

2.8 BONUS (non obligatoire donc)

Voici différents bonus possibles :

- Ajouter d'autres boutons à la calculatrice en leur choisissant une fonctionnalité, une position et une image spécifique.

- Ajouter d'autres fonctionnalités au menu de configuration, comme par exemple modifier la position d'un bouton, l'image d'un bouton, etc… et cela en cours d'exécution de la calculatrice. Dans ce cas, les différents boutons devraient également faire partie de l'« état » de la calculette et être enregistrés dans le fichier « config.dat ».

- Lorsque **l'exception CalculetteException** est lancée, actuellement les trois panneaux de la calculatrice sont affichés en rouge pendant une seconde. A la place, on pourrait imaginer qu'une autre fenêtre graphique s'ouvre (rappelez-vous qu'avec la librairie WindowSDL fournie, on ne peut ouvrir qu'une seule fenêtre à la fois) en contenant un message d'erreur adapté. Pour cela, il suffirait de créer une classe **FenetreErreur** héritant de **Fenetre** et ayant des variables membres adaptées.

- On pourrait imaginer une autre version de l'application dans laquelle le modèle MVC ne serait pas découpé en 3 classes distinctes Calculette – FenetreCalculette – Controleur. Une autre manière de faire est que la classe **FenetreCalculette** serait son propre Ecouteur (de boutons). Donc plus de classe Controleur. En fait, la classe **FenetreCalculette pourrait hériter de la classe Fenetre ET de la classe Ecouteur**. De plus, elle contiendrait une instance de la classe Calculette comme variable membre. Ainsi, le classe FenetreApplication réagirait elle-même aux clics des boutons (via la méthode ActionBouton) et pourrait agir directement sur l'objet calculette vu que ce serait une de ses variables membres. Comme vous le verrez, cette manière de faire est courante en Java.

Bon travail 😊 !