# 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.

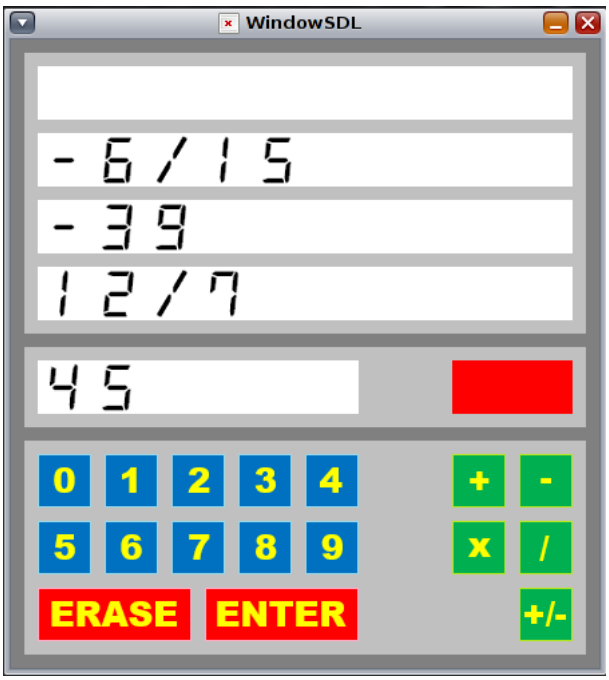
**1)** Chacun des membres d'une équipe d'étudiants doit être capable d'expliquer et de justifier l'intégralité du travail (pas seulement les parties du travail sur lesquelles il aurait plus particulièrement travaillé)

**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)

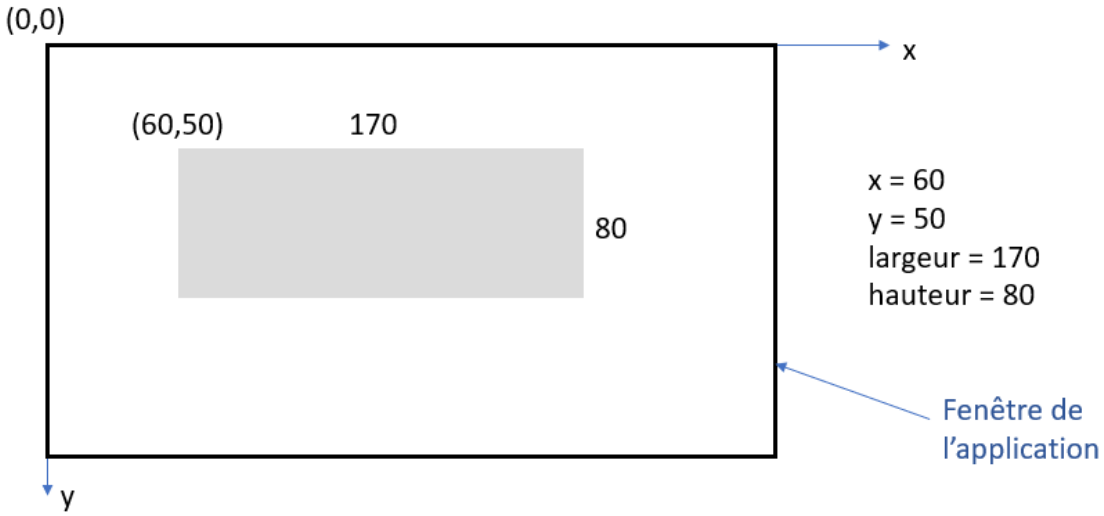
| Points principaux de l'évaluation | Subdivisions |
|---|---|
| • **Jeu de tests 1** : Implémentation d'une classe de base (**Panneau)** | Constructeurs, destructeurs |
| | Getters, Setters et méthode Affiche |
| | Fichiers Panneau.cpp, Panneau.h et makefile |
| • **Jeu de tests 2 :** Agrégation entre 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()

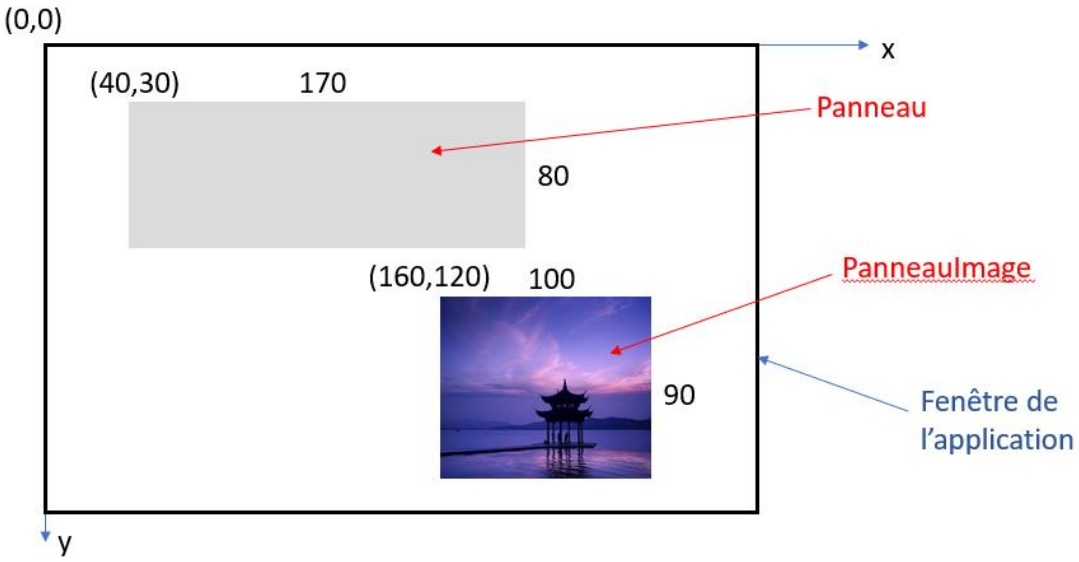
Un PanneauImage est un Panneau qui affichera une image à la place d'une zone de couleur uniforme. On vous demande donc de programmer la classe **PanneauImage**, héritant de la classe **Panneau** et qui :

- 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)]
```

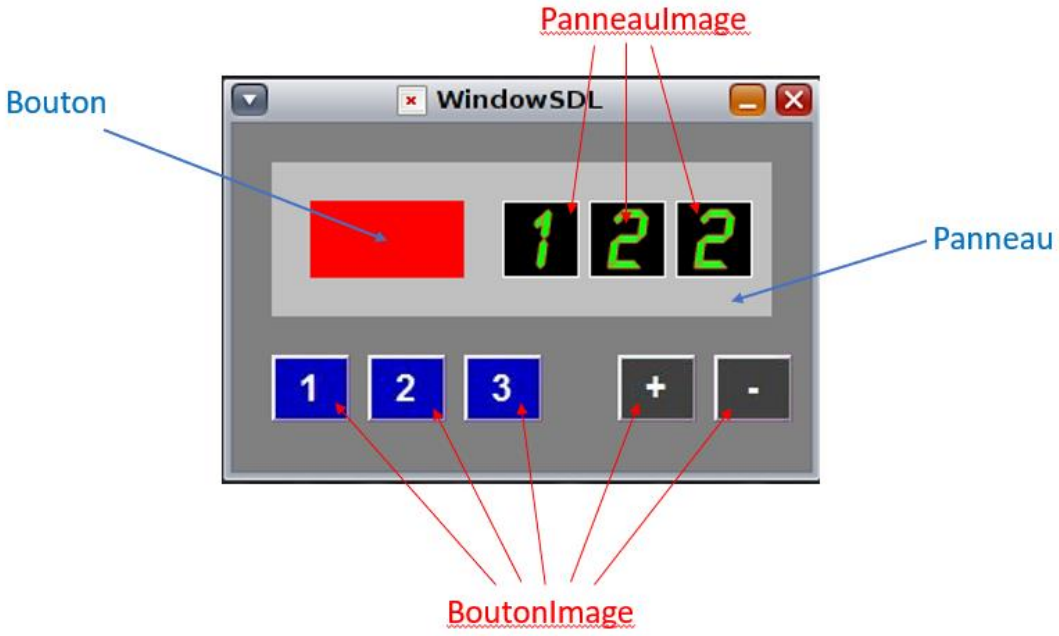
### c) La classe BoutonImage (héritage multiple) : Essai3()

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

To be continued ☺…