TP3: Héritage, dipôles et figures récursives

C++

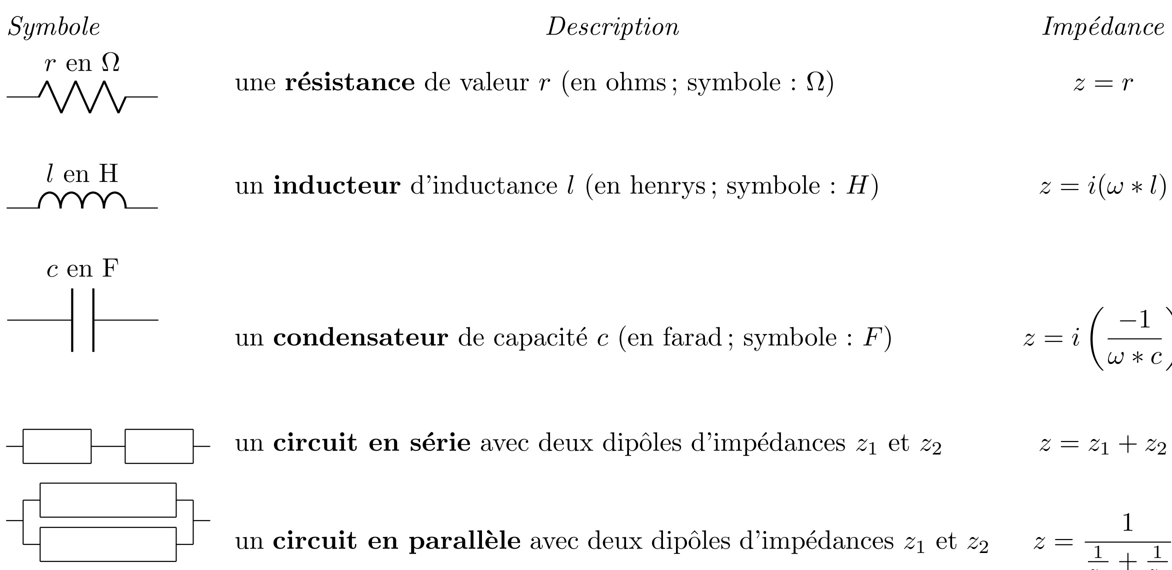
Objectifs pédagogiques :
- Savoir écrire une hiérarchie de classes d'après les consignes (Ex1).
- Concevoir une hiérarchie de classes pour résoudre un problème (Ex2).

À faire avant la séance :
— Tester l'exécution de l'exercice 2 (signaler les problèmes avant la séance).

1 Dipôles électriques

Les dipôles sont composés de composants "élémentaires" : *résistance*, *inducteur* et *capacité*. Ces composants élémentaires peuvent être combinés par un montage *en série* ou *en parallèle*. En fonction de ses composants, chaque circuit présente une résistance spécifique lorsqu'un voltage est appliqué. L'*impédence* étend cette notion de résistance aux courants alternatifs.

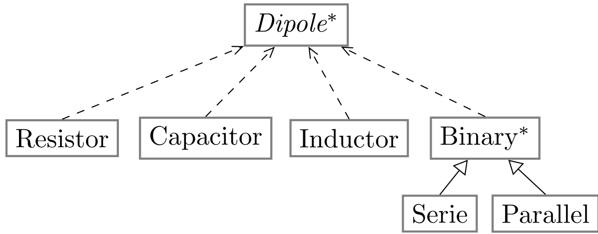
Soit ω un nombre réel appelé *pulsation* (*angular frequency* en anglais). *L'impédance* z d'un dipôle est un nombre complexe calculé de la façon suivante (seules les résistances ont une impédance dans $\mathbb{R}$).



Symbole	*Description*	*Impédance*
r en Ω	une **résistance** de valeur r (en ohms ; symbole : Ω)	$z = r$
l en H	un **inducteur** d'inductance l (en henrys ; symbole : H)	$z = i(\omega * l)$
c en F	un **condensateur** de capacité c (en farad ; symbole : F)	$z = i\left(\frac{-1}{\omega * c}\right)$
	un **circuit en série** avec deux dipôles d'impédances z_1 et z_2	$z = z_1 + z_2$
	un **circuit en parallèle** avec deux dipôles d'impédances z_1 et z_2	$z = \frac{1}{\frac{1}{z_1} + \frac{1}{z_2}}$

1.1 Modélisation

Nous allons modéliser les dipôles électriques en utilisant la hiérarchie de classes suivante :



`Dipole` définit une méthode `virtual std::complex<double> impedance(double omega) = 0` implementée dans les sous-classes. `omega` représente la pulsation pour laquelle il faut calculer l'impédance du circuit.

1.2 Implémentation de dipôles simples

Ouvrez maintenant le projet `dipoles/CMakeLists.txt` du code fourni dans `QtCreator` ou `VSCodium`.

▷ **Question 1:** Complétez l'implémentation dans `Resistor.cpp` et `Resistor.hpp` pour corriger les problèmes indiqués par le test `DipoleTest` concernant la classe.

▷ **Question 2:** Copiez les fichiers `Resistor.cpp` et `Resistor.hpp` pour écrire la classe `Inductor`, modifiez le fichier `CMakeLists.txt` pour ajouter l'implémentation à la bibliothèque, puis modifier la classe pour passer le test de `DipoleTest` associé à la classe. Décommentez le test de la classe `Inductor` dans le fichier `DipoleTest.cpp` pour rajouter les tests liés à la classe lors de l'exécution.

▷ **Question 3:** Implementez la classe `Capacitor` de façon similaire.

100Ω

($z = 100\ \Omega$)

FIGURE 1 – Résistance testée.

$7 \times 10^{-2}H$

($z \approx 22j\ \Omega$)

FIGURE 2 – Inducteur testé.

$10F$

($z \approx -0.0001j\ \Omega$)

FIGURE 3 – Capacité testée.

1.3 Circuits binaires

Nous allons maintenant implémenter les circuits binaires, en commençant par les montages en série.

▷ **Question 1:** Complétez la classe `Serie` afin de passer le test associé dans `DipoleTest` (à décommenter). Il correspond à la Figure 4 pour $\omega = 314$.

La classe `Serie` doit naturellement contenir des références vers les deux dipôles montés en série. Comme il peut s'agir indifféremment de résistances, inducteurs ou capacités, ces champs doivent être de type `Dipole&` ou `Dipole const&`. L'usage de références est indispensable car on ne peut pas écrire `Dipole d;` puisque cette classe est abstraite. Attention, ces références interdisent d'utiliser des objets temporaires comme paramètre du constructeur, car on risquerait de garder une référence sur un objet détruit entre temps. À la place, il faut utiliser des objets automatiques comme c'est fait dans le code qui teste votre code (la semaine prochaine, nous verrons une meilleure solution à ce problème : les pointeurs intelligents).

Les opérateurs de la classe `Serie` doivent naturellement déléguer le travail aux opérateurs de ces deux dipôles de la manière suivante

```
bool Serie::operator == (Serie const& other) const {
    return d1_==other.d1_ && d2_==other.d2_;
}
```

Pour cela, il faut que les opérateurs soient utilisables dans la classe Dipole alors que nous ne les avons utilisé que dans les classes filles pour l'instant. Le code que vous avez téléchargé permet déjà de définir le test d'égalité entre n'importe Dipoles et sous-classes. Observez comment le test d'égalité dans la super-classe vérifie que le type des deux opérandes correspond, tandis que le test d'égalité dans toutes les sous-classes commence par invoquer l'opérateur `==` de son ancêtre direct avant de vérifier l'égalité des champs spécifiques à la classe.

Écrire l'opérateur de sérialisation `<<` pour la classe `Dipole` s'avère plus compliqué car il n'est pas définit comme une méthode membre, mais dans une fonction séparée. Une solution est de définir une méthode abstraite pure dans `Dipole` avec le prototype suivant : `virtual void print_to(std::ostream& out) const = 0;` Cette méthode, définie dans toutes les sous-classes, écrit l'objet courant dans le flux en paramètre. Avec cela, on peut écrire simplement un opérateur de sérialisation pour la classe `Dipole` (marquée TODO dans le template). On peut alors supprimer les opérateurs de sérialisation des sous-classes.

▷ **Question 2:** Implémentez maintenant la classe `Parallel` de façon similaire à la classe `Serie`.

Décommentez le dernier test dans `DipoleTest` pour tester votre implémentation de `Parallel` sur le circuit de la Figure 5 pour $\omega = 314$.

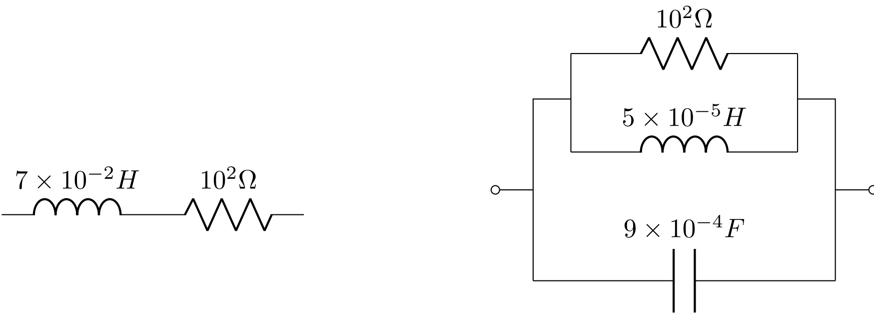


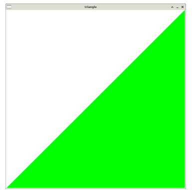
FIGURE 4 – Serie ($z \approx 100.0 + 21.98j\ \Omega$). FIGURE 5 – Parallel ($z \approx 0.2079 + -4.55j\ \Omega$).

▷ **Question 3:** Si vous avez suivi l'énoncé à la lettre, vous avez beaucoup de code dupliqué entre les classes `Serie` et `Parallel`. Factorisez ce code en introduisant une classe abstraite `Binary`. N'hésitez pas à modifier votre code pour le simplifier autant que possible.

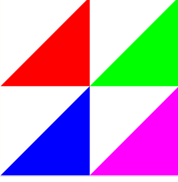
2 Figures récursives

L'objectif de cet exercice est de créer des figures géométriques composées de transformations simples.

Le code fourni donne trois classes dans le fichier `Picture.hpp` :



— `Picture` est une classe abstraite de quelque chose de dessinable.
— `Triangle` implémente cette classe, pour dessiner un triangle, comme à gauche.
— `FourPics` implémente également la classe `Picture` pour représenter quatre figures aux quatre angles, comme à droite où l'on combine des triangles de diverses couleurs.

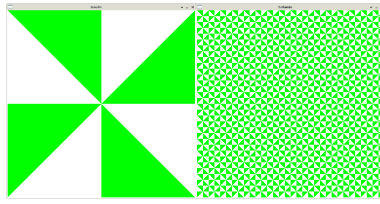


Le fichier `Main.cpp` constitue le programme principal. On y trouve une fonction `draw()` qui prend une `Picture` en paramètre et l'affiche dans une fenêtre ouverte pour l'occasion.

Vous remarquerez que les différents triangles sont créés comme des objets automatiques, sur la pile. Les paramètres à passer à leurs constructeurs sont donc passés entre accolades, et il faut passer leur adresse aux fonctions qui attendent des pointeurs.

L'objet de type `FourPics` est lui créé dans le tas, avec `new()`, principalement pour l'exemple. Cet objet n'est pas libéré à la fin du programme, ce qui poserait problème dans un vrai projet. Il n'est pas demandé de libérer la mémoire dans cet exercice. Une première solution serait de faire comme en C, en s'appliquant à avoir exactement un `delete()` par `new()`, mais c'est trop pénible en pratique. Une alternative serait de créer tous les objets sur la pile, mais les écritures sont assez vite disgracieuses dans les questions suivantes. Le mieux en C++11 serait d'utiliser les pointeurs intelligents (smart pointers), que nous verrons au prochain cours.

2.1 Mosaïque de rotations



L'objectif de cette question est d'abord de dessiner une sorte de moulin comme sur la figure de gauche ci-contre, puis de démultiplier cette figure 256 fois comme sur la figure la plus à droite.

Le moulin s'obtient naturellement en utilisant `FourPics` sur les quatre rotations d'un coin. Il faut donc définir une classe `Rotation`, qui applique une rotation à l'image passée en paramètre du constructeur.

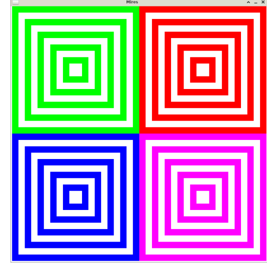
La fonction de dessin de cette classe `Rotation` doit être définie comme suit :

```
void draw(sf::RenderWindow* win, sf::Transform trans = sf::Transform()) const override  {
  p_.draw(win, sf::Transform(trans).rotate(angle_, window_size / 2, window_size / 2));
}
```

Le second paramètre de `draw()` est la transformation à appliquer à la figure. `sf::Transform(trans)` construit une copie du paramètre, ce qui est important car toutes les opérations définies par SFML sur la classe `sf::Transform` modifient le receveur. La rotation utilisée est appliquée par rapport au centre de la figure. Cette transformation est appliquée avant celles reçues en paramètre, et c'est ce qui permet de chaîner les transformations.

Une fois que la rotation fonctionne, on peut s'attaquer à la démultiplication par 256. On pourrait utiliser `FourPics` pour cela, mais il est plus simple de définir une classe `FourSame` permettant de représenter quatre fois la même image. Ne copiez pas `FourPics` pour définir `FourSame`, mais cherchez à réutiliser ce code.

2.2 Mosaïques de mires carrées

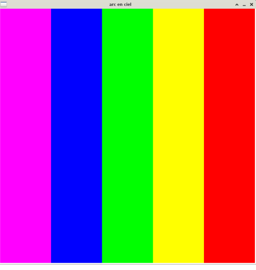


L'objectif est maintenant de dessiner la figure de droite. Elle est constituée de quatre mires de différentes couleurs. Chaque mire est une superposition de boîtes de plus en plus petites. Par exemple, la mire verte est une succession de boîtes vertes puis blanches dont la taille réduit de 10% à chaque fois.

Les deux briques principales à construire pour cette figure sont d'une part de pouvoir appliquer un ratio de réduction à la figure passée en paramètre, et d'autre part de pouvoir dessiner plusieurs figures ranger dans une collection. Il peut également être utile d'écrire une classe dessinant une mire de deux couleurs données pour factoriser du code.

Testez votre code étape par étape, et attachez-vous à le rendre aussi lisible que possible. Si vous avez besoin de commenter votre code pour cela, c'est que vous avez mal nommé vos classes et variables.

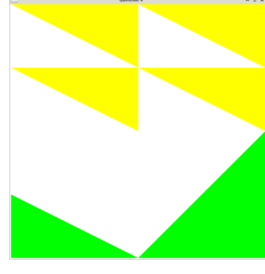
2.3 Arc en ciel



Pour dessiner l'arc-en-ciel ci-contre, vous aurez besoin de boîtes entièrement remplies (soit en faisant un `sf::ConvexShape` à 4 sommets, soit en utilisant directement `sf::RectangleShape`), et d'une collection de figures qui dessine ses différents éléments les à coté des autres, en les tassant avec une mise à l'échelle différente sur chaque axe. Les y sont laissés inchangés tandis que l'espace horizontal est partagé entre tous les éléments. Bien entendu, cette nouvelle collection devrait réutiliser le code de celle définie à la classe précédente, même si cela demande d'ajouter des accesseurs.

2.4 Miroirs

L'image ci-contre est une combinaison de triangles de différentes couleurs, dont certains sont inversés selon l'axe vertical et d'autres selon l'axe horizontal. On pourrait définir d'autres formes de base pour avoir les différents types de triangles, mais il est préférable de définir les transformations adéquates. Comme SFML ne dote pas ses transformations de méthode adéquate, il faut donner la matrice correspondante.

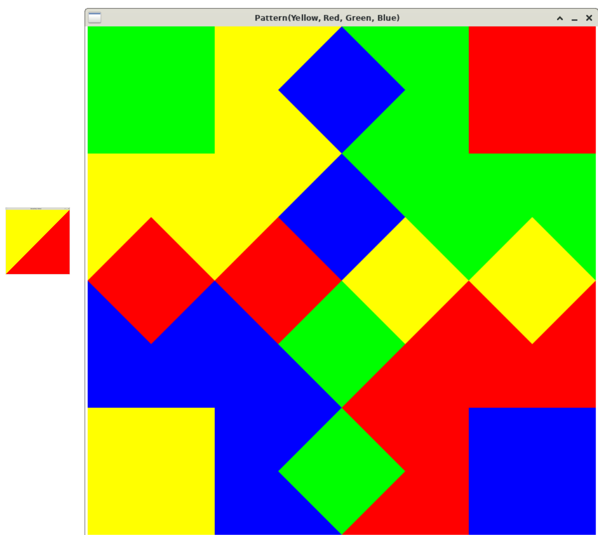


Le miroir vertical s'obtient avec la matrice $\begin{pmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$ et un décalage vers la droite, tandis que $\begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$ donne le miroir horizontal avec un décalage vers le haut.

Cette figure est construite avec `FourSame` appliqué aux triangles jaunes inversés. Vous aurez donc également besoin d'une collection verticale pour la figure finale. Comme d'habitude, attachez vous à la lisibilité de votre code, et factoriser ce qui peut l'être dans des classes ancêtres.

2.5 Figure plus avancée (optionnelle)

On souhaite maintenant faire la grande figure ci-dessous. C'est une spécialisation de `FourPics` dont 3 des sous-figures sont elles aussi des spécialisations de la même classe. La petite figure à gauche représente l'une des sous-sous-figure utilisée. C'est le sous-sous-cadrant III du le sous-cadrant III du cadrant II.



2.6 Faire plus joli

Si vous inventez de plus jolis motifs, n'oubliez pas de nous les faire partager. Vous pouvez aussi créer de nouvelles transformations et figures de bases pour cela. La beauté du code est tout aussi importante. Assurez vous d'avoir le code le plus lisible et le mieux factorisé possible. Il est cependant assez fastidieux de chercher à corriger les fuites mémoire, inéluctables ici si on veut un joli code source sans introduire de pointeurs intelligents.

▷ **Question 1:** Dessinez la hiérarchie de classes que vous avez réalisé, et discutez-la avec l'encadrante de TP.