TP4: Pointeurs intelligents, héritage et templates de classe
C++

Objectifs pédagogiques :
- Utiliser les pointeurs intelligents `std::unique_ptr`, `std::shared_ptr` (Ex 1, Ex 3) et `std::static_pointer_cast` (Ex 2)
- Implémenter un constructeur utilisant une `std::initializer_list` (Ex 1)
- Concevoir une classe avec template (Ex 2)
- Utiliser `using` pour définir des alias et simplifier le code (Ex 2)
- Concevoir des classes avec *header-only* (Ex 2)

1 Dipôles intelligents

La solution proposée la semaine dernière pour les dipôles électriques binaires (en série ou en parallèle) posait un problème de gestion de la mémoire. Comme nous ne souhaitions pas gérer explicitement la mémoire avec `new` et `delete` et que les circuits binaires utilisent des références vers leurs composants, les tests devaient utiliser des objets automatiques. Bien que fonctionnelle, cette solution n'est pas très robuste. Nous allons implémenter une solution avec des pointeurs intelligents.

1.1 Circuits binaires intelligents

▷ **Question 1:** Reprenez votre code de la séance précédente, et modifiez le afin de passer le nouveau cas de test `smart_test` proposé dans le code de cette semaine. Votre classe binaire devrait ressembler à ceci :

```
class SmartBinary : public Dipole {
protected:
    std::unique_ptr<Dipole> d1_;
    std::unique_ptr<Dipole> d2_;
public:
    SmartBinary(std::unique_ptr<Dipole> d1, std::unique_ptr<Dipole> d2);
    bool operator == (Dipole const& other) const;
};

```

Ne vous trompez pas en copiant le nouveau code de test : il suffit de copier `SmartBinaryTest.cpp` dans votre répertoire de la semaine passée, puis de modifier le `CMakeLists.txt` pour l'ajouter en tant que nouveau binaire de test.

1.2 Circuits n-aires intelligents (optionnel — en fin de séance)

Nous souhaitons maintenant implémenter les classes `NSerie` et `NParallel` permettant de connecter un nombre quelconque de dipôles, respectivement en série ou parallèle. Comme dans le cas binaire, l'impédance d'un circuit en série est la somme des impédances de ses composants tandis que l'impédance d'un circuit en parallèle est l'inverse de la somme des inverses des impédances de ses composants.

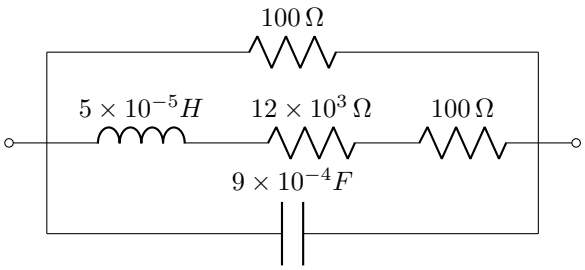
▷ **Question 1:** Implémentez les classes `NSerie`, `NParallel` ainsi que la classe abstraite `Nary` qui permet de factoriser du code entre `NSerie`, `NParallel`.

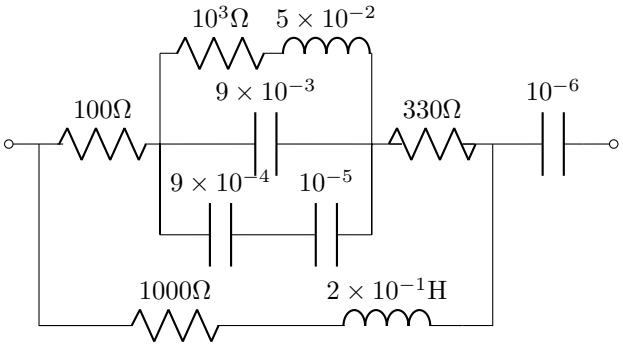
▷ **Question 2:** Écrivez un cas de test permettant de tester votre implémentation sur les circuits page suivante. Le plus simple pour instancier ces dipôles est probablement de doter vos classes d'un constructeur prenant un vecteur de pointeurs partagés vers des dipoles pour simplifier l'usage de ce constructeur.

```
1  // Prototype du constructeur:
2  explicit Nary(std::vector<std::shared_ptr<Dipole>> list);
3
4  // Dans le test:
5  auto serie = std::make_shared<NSerie>(std::vector<DipolePtr>({
6           std::make_shared<Inductor>(5e-5), std::make_shared<Resistor>(12e3)}));
```

Vous remarquerez que ce code utilise des pointeurs partagés `std::shared_ptr` à la place des pointeurs uniques de l'exercice précédent. C'est parce que le code utilisateur (lignes 5-6) utilise une liste entre accolades en paramètre du constructeur du vecteur. Cette écriture, nommée `std::initializer_list`, est une belle fonctionnalité C++11 pour simplifier le code utilisateur, mais le paramètre est forcément copié lors du passage. Il est donc impossible de l'utiliser pour des pointeurs uniques.

Les plus aventureux pourront tenter d'augmenter leur code pour ajouter un constructeur prenant directement un `std::initializer_list` au lieu d'utiliser ce vecteur temporaire passé en paramètre par une liste entre accolades directement.





2 Expressions

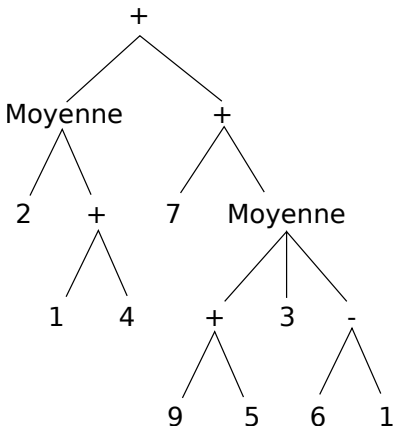
L'objectif de cet exercice est de définir un ensemble de classes permettant de représenter et de manipuler des expressions arithmétiques. Ces expressions sont constituées :

- d'opérandes constantes ou réelles,
- d'opérateurs binaires : $+$, $-$, $\times$ et $/$,
- d'opérateurs n-aires : Maximum, Moyenne, Somme, ...

Une expression peut être représentée sous la forme d'un arbre dont :

- la racine est l'opérateur le moins prioritaire,
- les nœuds (internes) sont des opérateurs,
- les feuilles sont les opérandes.

Par exemple, l'expression ci-dessous est représentée par l'arbre à droite.



$$\text{Moyenne}(2, 1+4) + (7 + \text{Moyenne}(9+5, 3, 6-1))$$

2.1 Première approche (sans héritage)

Nous allons d'abord essayer de résoudre le problème sans utiliser d'héritage. On utilisera une classe `FlatExpr` dotée d'un champ spécifique précisant le type d'expression. Par exemple, si ce champ vaut `'+'`, alors l'expression est une addition. Pour simplifier, on considérera uniquement le cas d'expressions constituées d'opérateurs binaires. Nous souhaitons pouvoir réaliser deux traitements sur une expression :

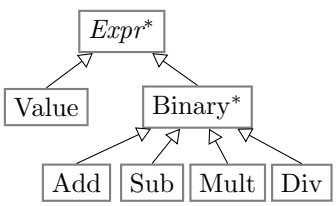
- `double FlatExpr::eval()` qui permet d'évaluer la valeur d'une expression (implémentée avec un `switch`). Par exemple, l'évaluation de l'expression représentée par l'arbre ci-dessus doit calculer la valeur `17.833...`
- `std::string FlatExpr::latex()` qui permet de représenter l'expression en LaTeX. On utilisera le symbole `\times` pour la multiplication, et on utilisera la macro `\frac` pour construire les fractions.
- `operator<<` permettant d'afficher (récursivement) un arbre dans un flux.

▷ **Question 1:** Implémentez une classe nommée `FlatExpr` permettant de passer le premier test du code fourni.

Bien que correcte, cette solution évolue très mal. Comme la logique est répartie entre plusieurs méthodes, l'ajout d'un nouvel opérateur demande de modifier plusieurs endroits du code. Cette façon de programmer porte un nom : il s'agit de shotgun surgery. Comme le nom l'indique, c'est une mauvaise habitude. On s'attachera donc dans la suite à implémenter une meilleure solution utilisant l'héritage.

2.2 Héritage, liaison dynamique et templates

L'arbre d'héritages de ce problème est assez similaire à celui du problème des dipoles. La principale différence est que nous allons écrire le code de façon générique de façon à pouvoir faire des opérations sur tous types de données (entiers, doubles, complexes).
Nous allons maintenant implémenter cette hiérarchie pour passer les tests fournis. Commencez par décommenter le second test du `CMakeLists.txt`.



Comme nous allons écrire beaucoup de petites classes, le plus simple est de mettre l'implémentation des méthodes directement dans le fichier d'entête afin de ne pas avoir à sauter d'un fichier à l'autre trop souvent. Il

n'y aura donc pas de `Expr.cpp`. Les bibliothèques implémentées de cette façon (comme `Catch2`, utilisée pour les tests en Prog2) sont dite *header-only*.

Le template fourni `Expr.hpp` est un peu complexe, mais vous avez tous les éléments pour le comprendre. La classe de base `Expr` est une template car on ne connait pas le type de retour de la méthode `eval()`. L'opérateur `<<` est défini directement dans la classe bien que ce ne soit pas une méthode de cette classe. C'est ce que signifie le `friend` devant le prototype, et c'est la solution la plus simple dans le cas d'un template de classe.

Utiliser `using` plutôt que `typedef` prend tout son sens ici, puisque notre alias de type est une template. Ce serait impossible avec un `typedef`.

▷ **Question 1:** Complétez la classe `Value` afin que son constructeur sauvegarde la valeur passée en paramètre, et que ses autres méthodes utilisent cette valeur.

▷ **Question 2:** Complétez la classe `Binary` afin que ses constructeurs sauvegardent les paramètres. Le constructeur recevant directement deux `T` doit créer des `Value` explicitement.

▷ **Question 3:** Complétez les classes `Add`, `Sub` et `Div` afin de pouvoir tester votre travail avec le test fourni.

▷ **Question 4:** Écrivez une classe `Mult` (pour la multiplication) sur le modèle des autres opérations. Décommentez le test fourni pour la tester.

▷ **Question 5:** Écrivez un cas de test pour vérifier que votre implémentation permet de faire des expressions de nombres complexes en utilisant `std::complex`.

▷ **Question 6: (optionnelle)** Intégrez votre implementation des nombres complexes du TP2, puis écrivez un test vérifiant qu'ils peuvent être utilisés comme base pour des `Expr`. Vous aurez probablement besoin d'ajouter des opérateurs à vos nombres complexes.

3 Figures récursives

Le code fourni la semaine dernière pour concevoir des figures récursives présentait des fuites mémoires. La mémoire allouée par les `new` dans le `Main.cpp` n'était jamais libérée.

▷ **Question 7:** Reprenez le code de la séance précédente et modifier le pour ne plus obtenir de fuites mémoires, grâce à des pointeurs intelligents. Tester votre programme avec Valgrind.