TD 4

Héritage

Ce TD permet d'appliquer la notion d'héritage avec deux courts exercices. Pour ceux qui sont en avance, le TD 5 peut être fait en remplacement du TD 4 et permet d'aller plus en profondeur, et d'explorer la notion de polymorphisme à travers la simulation d'un billard, ainsi que de mettre en œuvre un affichage graphique avec la bibliothèque SFML.

4.1 Échauffement : ajout de fonctionnalité par héritage

Considérons une classe `A` qui n'est pas sous votre contrôle (par exemple, une classe de la bibliothèque standard ou d'une bibliothèque externe). Nous voulons lui ajouter ou remplacer une fonctionnalité, une méthode par exemple, tout en gardant les autres fonctionnalités, sans changer son utilisation, et en évitant de tout ré-écrire.

Pour cela, nous pouvons créer une classe `B` héritant de la classe `A`, et simplement ajouter ou surcharger une méthode.

Dans les TDs précédents, nous avons écrit une classe `complexe`. En fait, la bibliothèque standard contient déjà une classe représentant les nombres complexes : `std::complex<TYPE>`, où `TYPE` est le type stockant les parties réelles et imaginaires (`float`, `double`...). On peut trouver sa documentation ici : `https://en.cppreference.com/w/cpp/numeric/complex`. Elle contient à peu près tout ce que l'on a définit dans les TDs précédents : norme, argument, opérations d'addition, de multiplication... Les parties réelles et imaginaires sont accessibles par des getters/setters `real` et `imag` respectivement.

Nous voudrions, pour l'exercice, ajouter une méthode normalisant un nombre complexe, c'est-à-dire modifiant le nombre de sorte à ce que sa norme vaille 1 ($z' = z/|z|$). Bien sûr, on pourrait simplement définir une fonction

```
void normalize (std::complexe<double>& z) {
  z = z / std::abs(z);
}
```

qui modifie son argument pour le normaliser (`normalize(z);`). Mais nous voulons plutôt une méthode pour cette opération (c'est-à-dire faire `z.normalize();`).

Créez une classe `complexe`, héritant de `std::complexe<double>`, et ajoutant une méthode
`void complexe::normalize();`
qui modifie les parties réelles et imaginaires (à travers les setters) pour normaliser le nombre.

Il vous faudra, comme toujours pour l'héritage, définir le constructeur (`complexe::complexe(double=0.,double=0.)`), le constructeur par copie (`complexe::complexe(const complexe&)`, où l'on pourra simplement relayer à celui de `std::complex<double>`), ainsi que l'opérateur d'affectation `complexe& complexe::operator=(const complexe&)`, que l'on pourra définir `= default;`). Il n'y a pas besoin de définir de destructeur, celui de la classe mère étant trivial. Comme d'habitude, testez votre code. Pour faire court, on pourra définir tout dans *main.cpp*, et le code ne devrait pas dépasser une cinquantaine de lignes.

C'est bien d'avoir un code qui fonctionne, mais c'est tout aussi important de comprendre *pourquoi* il fonctionne. Posez-vous les questions suivantes (pensez au polymorphisme, aux références, regardez les prototypes des fonctions et méthodes de la documentation...) :

1. Lorsque l'on fait `complexe z; std::cout << z;`, pourquoi cela fonctionne-t-il ? Nous n'avons pourtant pas défini de `operator<< (std::ostream&, complexe)` !
2. Lorsque l'on fait `complexe z; z *= 2;`, pourquoi cela fonctionne-t-il ? Nous n'avons pas défini de méthode pour la multiplication !
3. Pourquoi `z += ( std::complex<double>(0.25, 0) + 0.75 );` fonctionne ? Le résultat de l'addition est du type de la classe mère `std::complex<double>`, pas de la classe fille `complexe` !
4. D'où vient l'erreur suivante ?

```
TD4.cpp:23:25: error: conversion from ‘std::complex<double>’ to non-scalar type ‘complexe’ requested
 23 |         complexe z2 = z - complexe(1, 0);
    |                       ~~^~~~~~~~~~~~~~~
```

   Comment faire pour que cela fonctionne quand même ? Une seule ligne de code est nécessaire.

4.2 Réorganisation du code du TD 3

Dans le TD précédent, nous avons défini une classe `Réseau` pour représenter notre ensemble de sites. Mais tout le code relatif à la simulation du modèle d'Ising (initialisation, hamiltonien, algorithme de Métropolis, calcul d'observables...) a été défini en tant que fonctions normales.

Une autre organisation possible est la suivante : le code relatif au modèle d'Ising fait partie d'une classe `ModèleIsing` héritant de la classe `Réseau`, et il y a des méthodes plutôt que des fonctions. Cela a l'avantage, entre autres, de rendre le code plus modulaire et plus structuré.

Une autre façon de faire est la composition, où une classe `ModèleIsing` aurait un attribut de type `Réseau`, au lieu d'en hériter.

Les deux façons sont admissibles, mais pour l'exercice, nous allons ré-organiser le code en utilisant l'héritage. La figure 4.1 montre un diagramme des classes simplifié de la structure désirée. On pourrait encapsuler encore plus de logique dans la classe `ModèleIsing`, par exemple la boucle de simulation, le calcul du taux d'acceptation, le calcul automatique des observables et leur enregistement dans un fichier...

Le constructeur prenant (`nx`,`ny`) comme arguments initialisera le réseau avec des valeurs aléatoires, alors que celui prenant un troisième argument `uint8_t` initialisera un système uniforme, en prenant soin de vérifier que la valeur donnée est bien ± 1. Faites cet exercice de ré-organisation, et testez si votre code fonctionne comme précédemment.

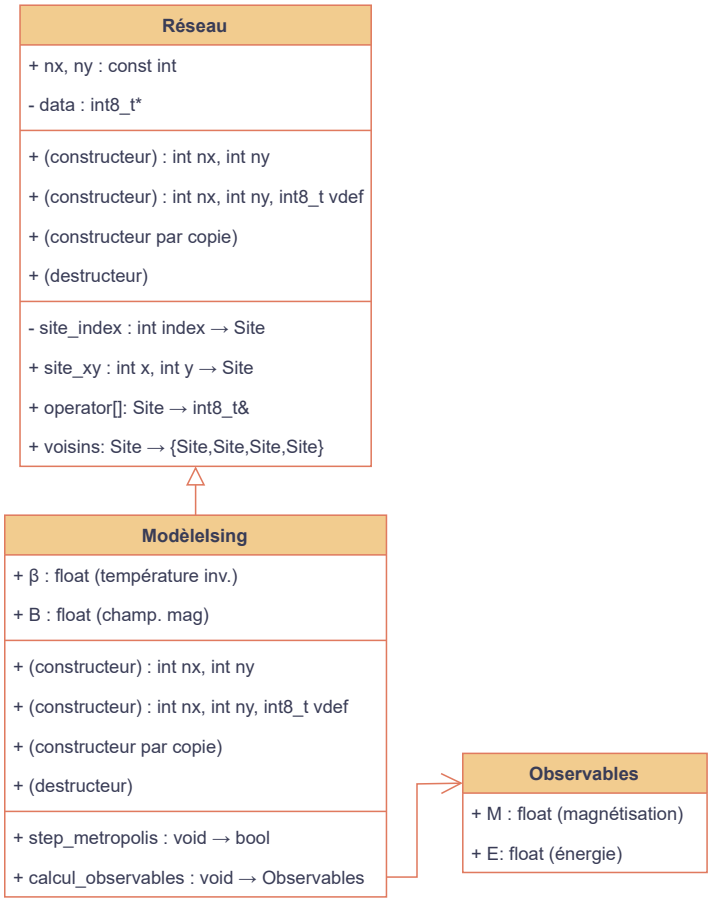

FIGURE 4.1 – Diagramme des classes simplifié.