POO & C++ : TP 2

EISE4 2014—2015

Pablo Rauzy

rauzy@enst·fr

pablo.rauzy.name/teaching.html#epu-cpp

3 octobre 2014

N'oubliez pas :

- Les TPs doivent être rendus par courriel à `rauzy@enst·fr` au plus tard le lendemain du jour où ils ont lieu avec `[EISE4]` suivi de vos noms dans le sujet du mail.
- Le code rendu doit être propre !
- Le TP doit être rendu dans une archive tar gzippée : `tar czvf NOMS.tgz NOMS` où `NOMS` est le nom du répertoire dans lequel il y a votre code (idéalement, les noms des gens du groupe) ; attention à ne pas rendre un répertoire avec des fichiers temporaires dus à la compilation (pensez à faire un `make clean` avant `tar`).
- Dans le répertoire avec vos noms, chaque exercice doit être dans son propre répertoire.
- Quand un exercice demande des réponses qui ne sont pas du code, vous les mettrez dans un fichier texte `reponses.txt` dans le répertoire de l'exercice.
- Le code est de préférence en anglais, avec des commentaires en français ou anglais, en restant cohérent.
- Le code doit être proprement indenté.
- Respectez les conventions de nommage :
  - variables : `nom_explicite` (*e.g.*, `game_type`),
  - fonctions : `verbeAction` (*e.g.*, `launchNewGame`),
  - classes : `NomExplicite` (*e.g.*, `BoardGame`),
  - attributs : comme variables + `_` (*e.g.* `max_number_of_player_`),
  - méthodes : comme fonctions,
  - accesseurs d'affectation : `set_` + nom de l'attribut sans le `_` final (*e.g.*, `set_max_number_of_player`),
  - accesseurs de consultation : nom de l'attribut sans le `_` final (*e.g.*, `max_number_of_player`),
  - constantes : `NOM_EXPLICITE` (*e.g.*, `VERSION_NUMBER`).
- Pensez à utiliser les outils de développement comme le debugger `gdb` ou le profiler `valgrind`.
- N'hésitez jamais à chercher de la documentation par vous-mêmes sur le net !

Dans ce TP :

- Surcharge des opérateurs.
- Héritage.

Exercice 0.

Récupération des fichiers nécessaires.

1. Pensez à organiser correctement votre espace de travail, par exemple tout ce qui se passe dans ce TP devrait être dans `~/cpp/tp2/`.

2. Récupérez les fichiers nécessaires au TP sur la page du cours.

3. Une fois que vous l'avez extrait de l'archive (`tar xzf tp2.tgz`), renommez le répertoire `tp2` en les noms de votre groupe (par exemple en `Dupont-Dupond`). Si vous ne le faites pas tout de suite, pensez à le faire avant de rendre votre TP.

Exercice 1.

Créer des classes correspondant à des grandeurs physiques (temps, distance, vitesse, ...), permet de manipuler des valeurs qui ont un sens, plutôt que de juste avoir des nombres. Dans ce cas, on peut se servir de la surcharge des opérateurs pour conserver la simplicité et la lisibilité de la manipulation de nombres, tout en gardant les avantages du typage de ces valeurs : le compilateur vérifie statiquement pour nous que le calcul est bien homogène.

Dans cet exercice, on va créer trois classes permettant de modéliser des calculs d'électrocinétique simples à base de la loi d'Ohm $U = R \cdot I$.

1. La classe `Volt` est simplement un conteneur pour une valeur de type `double` prise en paramètre optionnel par le constructeur (sinon ce sera zéro).

Pour l'instant, on veut simplement pouvoir :

- récupérer cette valeur avec un accesseur `val()`,
- écrire la valeur dans un stream avec l'opérateur `<<`.
- lire la valeur dans un stream avec l'opérateur `>>`.

→ Créez un fichier `volt.hpp` avec la déclaration de la classe `Volt`. Pensez à mettre une garde contre les inclusions récursives (`#ifndef VOLT_HPP___ ...`, inspirez vous des fichiers du premier TP si besoin).

2. → Créez un fichier `volt.cpp` avec l'implémentation de la classe `Volt`.

Lors de la surcharge de l'opérateur `<<` pensez à aussi afficher l'unité (`V` pour les Volts) après la valeur.

3. Il est important de tester son code régulièrement.

→ Créez un fichier exo1.cpp qui contiendra un `main` qui déclare un objet de type `Volt`, demande sa valeur à l'utilisateur puis l'affiche.

Utilisez le `Makefile` fourni pour la compilation.

4. → Faire la même chose pour une classe `Ohm` dans `ohm.hpp` et `ohm.cpp`. L'unité à afficher pour les ohms est Ω.

N'hésitez pas à utiliser le copier-coller, et testez la classe en mettant à jour le `Makefile` et le `main`.

5. → Idem pour la classe `Ampere` dans `ampere.hpp` et `ampere.cpp`.

6. Pour l'instant on ne peut pas du tout mélanger les valeurs de nos différentes unités.

→ Quelle erreur donne le compilateur si on essaye par exemple de multiplier un `Ampere` et un `Volt` ?

7. On veut autoriser certaines opérations, seulement celles qui conservent un sens physique. Pour l'exercice, on considère seulement la loi d'Ohm $U = R \cdot I$. Par exemple, on veut pouvoir multiplier un objet `Ohm` avec un objet `Ampere` et avoir en retour un objet `Volt`.

→ Ajoutez dans `ohm.hpp` et `ohm.cpp`, ce qu'il faut pour surcharger l'opérateur `*` comme on vient de le décrire.

Vous allez avoir besoin d'informer la classe `Ohm` de l'existence des classes `Volt` et `Ampere`. On peut déclarer l'existence d'une classe `Foo` sans donner plus de détails avec la déclaration `class Foo;`.

Pensez aussi à mettre à jour le `Makefile` avec les éventuelles nouvelles dépendances.

8. Modifiez maintenant le `main` pour demander à l'utilisateur une résistance en Ohms et une intensité en Ampères, puis retourner le nombre de Volts correspondant.

→ Que se passe-t-il si vous tapez `Volt v = i * r;` (où `i` est de type `Ampere` et `r` de type `Ohm`) ? Pourquoi ?

9. → Corrigez le problème.

10. On veut maintenant pouvoir aussi utiliser le fait que $I = U/R$ et que $R = U/I$.

→ Appliquez les changements nécessaires (y compris dans le `Makefile` si besoin) et utilisez les dans le `main`.

Exercice 2.

L'héritage permet comme on l'a vu en cours d'avoir une interface commune, un *sous-type* qui correspondra à une classe de base, dans plusieurs classes dérivées. Dans cet exercice on va mettre en œuvre cette fonctionnalité pour pouvoir créer des fonctions de manipulation de conteneurs, sans avoir à se soucier de leur implémentation.

1. On veut pouvoir travailler sur les éléments des conteneurs sans se soucier du type de conteneur (tableau, liste, arbre, tas, ...). Pour cet exercice, tous nos conteneurs manipuleront des entiers (`int`).

On veut disposer d'un "curseur" dans le conteneur et pouvoir :

- placer le curseur sur le premier élément ;
- savoir si le curseur est sur le premier élément ;
- placer le curseur sur le dernier élément ;
- savoir si le curseur est sur le dernier élément ;
- faire avancer ou reculer le curseur d'un élément ;
- lire ou écrire la valeur de l'élément sous le curseur ;

- savoir si le conteneur est vide ou pas.

L'implémentation de chacune de ces opérations dépend du type de conteneur, on va donc en faire des fonctions virtuelles pures dans l'interface commune.

L'interface offerte par ces fonction permet de créer des traitements génériques. Deux exemples sont donnés.

- Le premier, `iter`, permet d'appliquer une fonction à chaque élément sans récupérer son résultat (la fonction doit prendre un `int` et retourne `void`). Cela peut par exemple servir à afficher les éléments.
- Le second, `map`, permet de faire la même chose mais cette fois en utilisant le résultat de la fonction (qui doit du coup prendre un `int` et en renvoyer un autre) pour remplacer la valeur de chaque élément. Cela peut par exemple servir à ajouter 1 à chaque élément.

→ Regardez le contenu des fichiers `iterable.hpp` et `iterable.cpp`, et vérifiez qu'il correspond bien à ce que l'on souhaite.

2. Maintenant, nous allons créer un conteneur de vecteurs (tableaux de taille fixe).

→ Ouvrez les fichiers `vector.hpp` et `vector.cpp` lisez le code puis ajoutez une surcharge de l'opérateur `[]` qui permet de lire et écrire dans le vecteur, avec vérification des bornes.

3. Avant d'aller plus loin, créez un fichier `main.cpp` pour tester votre classe `Vector`, et un `Makefile` pour compiler les deux modules (`main`, `vector`).

→ Compilez et testez votre code.

4. → Dans `vector.hpp`, faites hériter `Vector` de `Iterable` et justifiez votre choix de politique de dérivation.

5. Ajoutez ce qu'il faut dans le `Makefile` pour prendre en compte le module `Iterable`.

→ Recompilez le programme. Quelle erreur obtenez vous ? Pourquoi ?

6. → Déclarez dans `vector.hpp` les fonctions virtuelles pures de `Iterable` et redéfinissez-les dans `vector.cpp`. Vérifiez que vous n'avez plus de problème de compilation.

7. → Ajoutez dans le fichier `main.cpp` de quoi tester que chaque fonction membre de la classe `Vector` marche correctement (y compris les fonctions héritées) et expliquez soit à l'écrit soit en commentaire dans le `main` comment vous avez testé chaque fonction et pourquoi votre test est suffisant.

8. Occupons-nous maintenant d'un autre conteneur.

Un conteneur de liste itérable vous est fourni, mais il n'est pas complet. Ouvrez les fichiers `list.hpp` et `list.cpp`.

→ Lisez-les attentivement, et expliquez le fonctionnement de la classe `List`, son interaction avec la classe `Node` et le rôle de celle-ci.

9. → Complétez les fonctions qui ont besoin de l'être (voir les commentaires dans le code).

10. → Ajoutez dans le fichier `main.cpp` de quoi tester que chaque fonction membre de la classe `List` marche correctement (y compris les fonctions héritées) et expliquez soit à l'écrit soit en commentaire dans le `main` comment vous avez testé chaque fonction et pourquoi votre test est suffisant.

Mettez évidemment à jour le `Makefile` pour prendre un compte le module `List`.

11. Constatez que les mêmes fonctions `iter` et `map` de la classe `Iterable` sont utilisables indifféremment sur des `List` ou sur des `Vector`.

→ Écrivez vous-même une nouvelle fonction dans le fichier `main.cpp` qui utilise l'interface `Iterable` pour itérer sur les éléments en ordre inverse et n'afficher que les multiples d'un nombre qu'on lui a passé en argument, puis testez-la pour les `List` et pour les `Vector`.