Cours MOOC EPFL d'introduction à la programmation orientée objet, illustré en C++

Cinquième et dernier devoir
Synthèse

J.-C. Chappelier & J. Sam

1 Exercice 1 — Cuisine

Le but de cet exercice est de développer quelques fonctionnalités pour gérer des recettes de cuisine.

1.1 Description

Télécharger le programme `restaurant.cc` fourni et le compléter suivant les instructions données ci-dessous.

ATTENTION : vous ne devez en aucun cas modifier ni le début ni la fin du programme fourni, juste ajouter vos propres lignes à l'endroit indiqué. Il est donc impératif de respecter la procédure suivante :

1. sauvegarder le fichier téléchargé sous le nom `restaurant.cc` ou `restaurant.cpp` ;
2. écrire le code à fournir (voir ci-dessous) entre ces deux commentaires :

```
/*****************************************
 * Complétez le programme à partir d'ici.
 *****************************************/

/*****************************************
 * Ne rien modifier après cette ligne.
 *****************************************/
```

3. sauvegarder et tester son programme pour être sûr(e) qu'il fonctionne correctement, par exemple avec les valeurs utilisées dans l'exemple de déroulement donné plus bas ;

4. soumettre le fichier modifié (toujours `restaurant.cc` ou `restaurant.cpp`) dans « My submission » puis « Create submission ».

Le code fourni crée un recette sur la base de produits, l'affiche, y trouve la quantité totale d'un produit donné et l'adapte à une quantité différente.

Un exemple de déroulement possible est fourni plus bas.

1.2 Classes à produire

Une *recette* est constituée d'une liste d'*ingrédients*. Elle a un nom et est aussi caractérisée par le nombre de fois que l'on souhaite la réaliser (un `double`, on peut faire une fois et demi la recette par exemple). Nous appellerons cet attribut `nbFois_` dans ce qui suit.

Un ingrédient est constitué :

— d'un *produit* ;
— d'une quantité (de ce produit, un `double`).

Un produit est caractérisé par :

— son nom (une chaîne de caractères, `"beurre"` par exemple) ;
— l'unité utilisée usuellement pour ce produit (une chaîne de caractères, `"grammes"` par exemple).

Il peut être *cuisiné* c'est-à-dire réalisé à partir d'une recette. Un produit cuisiné sera donc caractérisé par sa recette.

Il vous est demandé de coder les classes : `Produit`, `Ingredient`, `Recette` et `ProduitCuisine` correspondant à la description précédente.

Toutes les listes seront modélisées au moyen de `vector`. Les ajouts dans ces listes se feront toujours **en fin de liste**.

Vous n'ajouterez aucun constructeur (ou opérateur d'affectation) en dehors de ceux qui vous sont explicitement demandés.

La classe `Produit` Les méthodes publiques de la classe `Produit` seront pour commencer :

— un constructeur permettant d'initialiser le nom et l'unité du produit au moyen de valeurs passées en paramètre (dans cet ordre) ; par défaut l'unité est la chaîne de caractères vide ;

— les getters `getNom()` et `getUnite()` ;

— une méthode `toString()` retournant simplement le nom du `Produit`, sans aucun ajout ni modification.

La classe `Ingredient` Les méthodes publiques de la classe `Ingredient` seront :

— un constructeur permettant d'initialiser le produit et sa quantité au moyen de valeurs passées en paramètres dans cet ordre (le produit sera initialisé au moyen d'une référence sur le produit passée en paramètre, sans copie) ;

— les getters `getProduit()` et `getQuantite()` ;

NOTE : `getProduit` doit retourner une référence constante au produit.

— une méthode `descriptionAdaptee()` produisant une représentation de l'`Ingredient` respectant ***strictement*** le format suivant :
« `<quantite> <unite> de <representation_produit>` »
où `<representation_produit>` est la chaîne de caractères représentant le produit dont les quantités auront été adaptées à la quantité de l'ingrédient. Vous pourrez utiliser la méthode `toString()` de la classe `Produit` pour produire `<representation_produit>`. `<unite>` est l'unité associée au produit.
NOTE : Pour implémenter `descriptionAdaptee()` , vous aurez à utiliser la méthode `adapter` telle que décrite un peu plus loin pour les produits.

La classe `Recette` Les méthodes publiques de la classe `Recette` seront pour commencer :

— un constructeur initialisant le nom et le nombre de fois qu'est réalisée la recette au moyen de valeurs passées en paramètre (dans cet ordre) ; par défaut, le nombre de fois vaut `1.` ;

— une méthode `void ajouter(const Produit& p, double quantite)` ajoutant un ingrédient construit au moyen des paramètres à la liste des ingrédients de la recette. La quantité de l'ingrédient vaudra `nbFois_` multiplié par la quantité passée en paramètre ;

— une méthode `Recette adapter(double n)` retournant une nouvelle recette correspondant à la recette courante réalisée `n` fois (sont attribut `nbFois_` vaudra celui de la recette courante multiplié par `n`).
Notez ici que les ingrédients de la recette courante ont déjà vu leur quantité multipliée par son `nbFois_` (voir méthode précédente) ; il faudra donc tenir compte de ce facteur dans le calcul des quantités de la nouvelle recette ;

— une méthode `toString()` produisant une représentation d'une `Recette` respectant ***strictement*** le format suivant :

```
Recette "<nom>" x <nb_fois>:
1. <ingredient_1>
2. <ingredient_2>
...
```

où `<nb_fois>` est la valeur de l'attribut `nbFois_` et `<ingredient_i>` est la chaîne de caractères représentant le i[e] ingrédient de la recette [1]. Voir l'exemple de déroulement plus bas pour plus de détails.
Attention : le dernier ingrédient **ne** devra **pas** être suivi d'un retour à la ligne ! On suppose que toutes les recettes ont au moins un ingrédient.

La classe `ProduitCuisine` Un `ProduitCuisine` est un `Produit`.

Les méthodes publiques de la classe `ProduitCuisine` seront :

— un constructeur conforme à la méthode `main()` fournie et permettant d'initialiser le nom du produit cuisiné au moyen d'une valeur passée en paramètre ; l'unité vaudra toujours `"portion(s)"` ; le nom de la recette du produit cuisiné sera le même que celui du produit ;

— une méthode `void ajouterARecette(const Produit& produit, double quantite)` permettant d'ajouter un ingrédient à la recette du produit (vous utiliserez la méthode `ajouter` de la classe `Recette`) ;

— une méthode `const ProduitCuisine* ProduitCuisine adapter(double n)` retournant un pointeur sur un nouveau produit cuisiné correspondant au produit courant dont la recette est adaptée `n` fois ;

— une méthode `toString()` produisant une représentation d'un `ProduitCuisine` en respectant ***strictement*** le format suivant :

```
<produit>
<recette>
```

où `<produit>` correspond à la représentation traditionnelle d'un produit [2] et `<recette>` correspond à la représentation de la recette du produit [3]. Voir l'exemple de déroulement plus bas pour plus de détails.
Attention : la représentation du produit cuisiné **ne** devra **pas** être suivie d'un retour à la ligne !

1. voir la classe `Ingredient` et sa méthode `descriptionAdaptee()`
2. voir la classe `Produit` et sa méthode `toString()`
3. voir la classe `Recette` et sa méthode `toString()`

Le produit d'un ingrédient peut désormais être soit un produit (de base) soit un produit cuisiné. La méthode `toString()` de l'ingrédient devra utiliser en guise de `<representation_produit>` celle du produit dont la recette a été adaptée à la quantité.

Les méthodes `adapter, toString` se doivent donc d'être *polymorphique* pour les produits.

Retouchez donc les classes `Produit`, `ProduitCuisine` en y ajoutant/changeant les méthodes `adapter, toString` nécessaires.

Comme il n'y a rien à adapter pour un produit de base (pas de recette pour le produit dont il faudrait adapter les quantités) ; on se contentera de retourner l'objet courant (`this`).

Le type de retour de `Produit::adapter()` devra être `const Produit*`.

NOTE : Les méthodes `adapter` des produits ne seront utilisées ici que pour produire les représentations adaptées aux quantités pour les ingrédients (méthode `descriptionAdaptee()`). On supposera néanmoins que l'on souhaite aussi utiliser `adapter` dans d'autres contextes (cette fonctionnalité sera d'ailleurs testée séparément par le correcteur).

Méthode `quantiteTotale` On souhaite finalement permettre de rechercher la quantité totale d'un produit de nom donné dans une recette. La méthode publique
`double quantiteTotale(const string& nomProduit)`
de la classe `Recette` cherchera le produit en question dans la liste de ses ingrédients (lequels peuvent être des produits cuisinés contenant aussi le produit recherché). Vous programmerez pour réaliser ce traitement des méthodes
`double quantiteTotale(const string& nomProduit)` dans :

- la classe `Recette` : où elle retournera la somme des quantités totales de ce produit trouvées dans chaque ingrédient ;
- la classe `Produit` : où elle retournera `1.` si le produit a pour nom `nomProduit` et `0.` sinon ;
- la classe `ProduitCuisine` : où elle retournera `1.` si le produit a pour nom `nomProduit`, et sinon, la quantité totale de ce produit dans la recette du produit cuisiné, cette méthode doit re-implémenter celle définie dans `Produit` ;
- la classe `Ingredient` : où elle retournera la quantité de l'ingrédient multipliée par la quantité totale du produit recherché dans le produit de l'ingrédient.

Il vous est donc demandé de coder la hiérarchie de classes et les différentes fonctionnalités découlant de cette description. Vous éviterez la duplication de code, le masquage d'attributs, de méthodes et nommerez les classes tel qu'il vous est suggéré de le faire dans l'énoncé.

1.3 Exemple de déroulement

```
glaçage au chocolat
  Recette "glaçage au chocolat" x 1:
  1. 200.000000 grammes de chocolat noir
  2. 25.000000 grammes de beurre
  3. 100.000000 grammes de sucre glace
glaçage au chocolat parfumé
  Recette "glaçage au chocolat parfumé" x 1:
  1. 2.000000 gouttes de extrait d'amandes
  2. 1.000000 portion(s) de glaçage au chocolat
  Recette "glaçage au chocolat" x 1:
  1. 200.000000 grammes de chocolat noir
  2. 25.000000 grammes de beurre
  3. 100.000000 grammes de sucre glace
===  Recette finale  =====
  Recette "tourte glacée au chocolat" x 1:
  1. 5.000000  de oeufs
  2. 150.000000 grammes de farine
  3. 100.000000 grammes de beurre
  4. 50.000000 grammes de amandes moulues
  5. 2.000000 portion(s) de glaçage au chocolat parfumé
  Recette "glaçage au chocolat parfumé" x 2:
  1. 4.000000 gouttes de extrait d'amandes
  2. 2.000000 portion(s) de glaçage au chocolat
  Recette "glaçage au chocolat" x 2:
  1. 400.000000 grammes de chocolat noir
  2. 50.000000 grammes de beurre
  3. 200.000000 grammes de sucre glace
Cette recette contient 150 grammes de beurre

===  Recette finale x 2 ===
  Recette "tourte glacée au chocolat" x 2:
  1. 10.000000  de oeufs
  2. 300.000000 grammes de farine
  3. 200.000000 grammes de beurre
  4. 100.000000 grammes de amandes moulues
```

```
  5. 4.000000 portion(s) de glaçage au chocolat parfumé
  Recette "glaçage au chocolat parfumé" x 4:
  1. 8.000000 gouttes de extrait d'amandes
  2. 4.000000 portion(s) de glaçage au chocolat
  Recette "glaçage au chocolat" x 4:
  1. 800.000000 grammes de chocolat noir
  2. 100.000000 grammes de beurre
  3. 400.000000 grammes de sucre glace
Cette recette contient 300 grammes de beurre
Cette recette contient 10  de oeufs
Cette recette contient 8 gouttes de extrait d'amandes
Cette recette contient 4 portion(s) de glaçage au chocolat

==========================

Vérification que le glaçage n'a pas été modifié :
glaçage au chocolat
  Recette "glaçage au chocolat" x 1:
  1. 200.000000 grammes de chocolat noir
  2. 25.000000 grammes de beurre
  3. 100.000000 grammes de sucre glace
```

2 Exercice 2 — Bataille navale

Nous nous intéressons dans cet exercice à modéliser de façon élémentaire un jeu de bataille navale avec des *navires*. Ces derniers pourront être des navires *pirates*, des navires *marchands* ou des navires *félons*, à la fois marchands et pirates.

2.1 Description

Télécharger le programme `bateaux.cc` fourni et le compléter suivant les instructions données ci-dessous.

ATTENTION : vous ne devez en aucun cas modifier ni le début ni la fin du programme fourni, juste ajouter vos propres lignes à l'endroit indiqué. Il est donc impératif de respecter la procédure suivante :

1. sauvegarder le fichier téléchargé sous le nom `bateaux.cc` ou `bateaux.cpp` ;
2. écrire le code à fournir (voir ci-dessous) entre ces deux commentaires :

```
/*****************************************
 * Complétez le programme à partir d'ici.
```

```
 ******************************************/

/******************************************
 * Ne rien modifier après cette ligne.
 ******************************************/
```

3. sauvegarder et tester son programme pour être sûr(e) qu'il fonctionne correctement, par exemple avec les valeurs utilisées dans l'exemple de déroulement donné plus bas ;
4. soumettre le fichier modifié (toujours `bateaux.cc` ou `bateaux.cpp`) dans « My submission » puis « Create submission ».

Le code fourni déjà quelques éléments utilitaires et un `main()` composés de trois parties correspondant aux sections de la présente description.

Parmi les éléments déjà fournis :

— un type énuméré représentant les différents pavillons possibles (certains diraient « drapeaux », à tort) ;
— un type énuméré représentant les différents états possibles pour un navire : intact, endommagé ou coulé ;
— une fonction utilitaire `sq(int)` retournant le carré d'un nombre entier reçu en paramètre ;
— une classe `Coordonnees` permettant de représenter des coordonnées entières à deux dimensions (typiquement dans une grille imaginaire) ;
— un programme principal auquel votre code devra *strictement* se conformer ; utilisez-le comme source d'inspiration de vos classes et méthodes.

Le code que vous devez fournir est décrit dans les sections suivantes. Il devra être proprement encapsulé et éviter la duplication de code.

2.2 Compléter les outils de base fournis

Dans la partie autorisée, après le commentaire

```
/*************************************************
 * Compléter le code à partir d'ici
 *************************************************/
```

et après l'accolade fermant la classe `Navire`, commencez par développer les éléments suivants :

— l'opérateur `+=` pour la classe `Coordonnees` permettant d'ajouter des coordonnées entre elles (cela sera utile pour déplacer les navires) ; il suffit ici d'ajouter le `x` des coordonnées reçues au `x` de l'instance courante et procéder de même pour les `y` ;

— définition d'une fonction `distance` retournant (sous forme d'un `double`) la distance entre deux `Coordonnees` ; la distance entre deux coordonnées (x_1, y_1) et (x_2, y_2) se calcule selon la formule

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}\,;$$

— surcharge de l'opérateur d'affichage `<<` pour les `Coordonnees` ; le format d'affichage à respecter est le suivant : « `(<x>, <y>)` », où `<x>` est la valeur de la première coordonnée et `<y>` celle de la seconde ; par exemple :
`(1, 2)`

— surcharge de l'opérateur d'affichage `<<` pour les `Pavillons`, qui affichera :
  — `"pirate"` pour la valeur `JollyRogers` ;
  — `"français"` pour la valeur `CompagnieDuSenegal` ;
  — `"autrichien"` pour la valeur `CompagnieDOstende` (Ostende est bien en Belgique de nos jours, mais à l'époque elle était sous occupation autrichienne !) ;
  — et `"pavillon inconnu"` dans les autres cas ;

— surcharge de l'opérateur d'affichage `<<` pour les `Etats`, qui affichera :
  — `"intact"` pour la valeur `Intact` ;
  — `"ayant subi des dommages"` pour la valeur `Endommage` ;
  — `"coulé"` pour la valeur `Coule` ;
  — et `"état inconnu"` dans les autres cas.

2.3 La classe `Navire`

Un `Navire` est caractérisé par :

— des coordonnées (cf classe fournie) permettant de repérer sa position sur une grille ; nommez cet attribut `position_` (avec le souligné à la fin) ;
— un *pavillon* (type énuméré fourni, nommé `pavillon_`) indiquant à quel compagnie appartient le navire ;
— une information `etat_` indiquant son état (au moyen du type énuméré `Etat` fourni).

La classe `Navire` comportera par ailleurs :

— un constructeur initialisant les coordonnées et le pavillon du navire au moyen de valeurs passées en paramètre (voir la méthode `main()` fournie) ; le navire « construit » sera intact (!) ;
— une méthode `position()` retournant les coordonnées du navire ;
— une méthode `void avancer(int de_x, int de_y)` permettant, si le navire n'est pas coulé, de le déplacer de `de_x` unités horizontalement

et de `de_y` unités verticalement (`de_x` et `de_y` peuvent être négatifs) ; utiliser pour cela l'opérateur `+=` des `Coordonnees` ;
- une méthode `void renflouer()` permettant de remettre le navire dans l'état `Intact` ;
- une méthode `afficher` permettant d'afficher le navire comme indiqué dans l'exemple de déroulement plus bas, typiquement au format :
  `<nom générique> en (<x>, <y>) battant pavillon <pavillon>, <etat>`
  (sur une seule ligne) où `<nom générique>` est le genre du navire (« bateau pirate », « navire marchand » ou « navire félon »), `<pavillon>`, la valeur de son pavillon, `<x>` et `<y>`, ses coordonnées et `<etat>` son état, comme indiqué plus haut.

Ajoutez également une fonction `distance()` retournant la distance entre deux `Navires`, ainsi qu'une surcharge de l'opérateur d'affichage `<<` pour les navires (produisant la même sortie que `afficher`).

Ajoutez enfin à la classe `Navire`, un attribut de classe, constant, nommé `rayon_rencontre` et valant 10.

2.4 `Pirate`, `Marchand` et `Felon`

Parmi les navires, nous avons des navires pirates (classe `Pirate`), des navires marchands (classe `Marchand`) et des navires félons (classe `Felon`), à la fois marchands et pirates . Ces classes seront dotées de constructeurs conformes au `main()` fourni. Pour les navires félons, ceux-ci devront avoir *un* seul état, *une* seule position, *un* seul pavillon, tout en évitant au maximum la duplication de code (plus de précisions ci-dessous lors de l'explication des combats, vous pouvez donc y revenir plus tard).

Pour tester les classes `Pirate` et `Marchand`, utilisez la partie « `Test de la partie 1` » du `main()` fourni (cf exemple de déroulement fourni plus bas).

Mauvaises rencontres Pour simuler le jeu de bataille navale, on adopte le critère suivant : si deux navires, non coulés, ont des pavillons différents et sont à une distance inférieure à `Navire::rayon_rencontre` (définie plus haut), alors ils se confrontent, sinon rien ne se passe. Il vous est donc demandé de doter vos classes

- d'une méthode `attaque()` prenant un autre navire en paramètre, et qui nous servira à décrire comment un navire en attaque un autre ;
- d'une méthode `replique()` prenant un autre navire en paramètre, et qui nous servira à décrire comment un navire réplique s'il est attaqué par un

autre ;
- d'une méthode `est_touche()` ne prenant aucun paramètre, et qui nous servira à décrire ce qui se passe si un navire est touché ;
- et enfin une méthode `rencontrer()`, prenant un autre navire en paramètre, et qui gère la rencontre d'un *autre* navire en :
  - testant si les conditions décrite plus haut sont remplies,
  - si elles le sont, appelant la méthode `attaque` puis la méthode `replique` (en passant l'autre navire en argument).

Au niveau des `Navire` (en général), on ne sait pas définir les méthodes `attaque()`, `replique()`, ni `est_touche()` ; et l'on souhaite de plus imposer leur définitions dans toutes les versions spécialisées des navires.

Au niveau des navires spécifiques, les règles de confrontation sont les suivantes :

1. si un navire pirate `b1` attaque un autre navire `b2`, il « crie » (= affiche) « `A l'abordage !` » et `b2` est touché (méthode `est_touche`) ;
2. si un navire pirate `b1` réplique à un autre navire `b2`, il affiche « `Non mais, ils nous attaquent ! On riposte !!` » et il attaque `b2` ;
3. si un navire pirate est touché, son état diminue d'un niveau : il passe de intact à endommagé ou de endommagé à coulé :
4. si un navire marchand `b1` attaque un autre navire `b2`, il lui « crie » des insultes : " `On vous aura ! (insultes)` " et rien d'autre ne se passe ;
5. si un navire marchand `b1` réplique à un autre navire `b2`, il affiche « `SOS je coule !` » si son état est « coulé » et sinon il affiche « `Même pas peur !` » ;
6. si un navire marchand est touché, il coule ;
7. les navires félons
   - attaquent comme les pirates ;
   - ripostent comme les marchands ;
   - sont touché comme les pirates.

Différents scénarios de rencontres peuvent être testés à l'aide de la partie « `Test de la partie 2` » du `main()` fourni (cf exemple de déroulement fourni plus bas).

2.5 Exemple de déroulement

```
===== Test de la partie 1 =====

bateau pirate en (0, 0) battant pavillon pirate, intact
```

```
navire marchand en (25, 0) battant pavillon français, intact
Distance : 25
Quelques déplacements...
  en haut à droite :
bateau pirate en (75, 10) battant pavillon pirate, intact
  vers le bas :
bateau pirate en (75, 5) battant pavillon pirate, intact

===== Test de la partie 2 =====

Bateau pirate et marchand ennemis (trop loins) :
Avant la rencontre :
bateau pirate en (75, 5) battant pavillon pirate, intact
navire marchand en (25, 0) battant pavillon français, intact
Distance : 50.2494
Apres la rencontre :
bateau pirate en (75, 5) battant pavillon pirate, intact
navire marchand en (25, 0) battant pavillon français, intact

Bateau pirate et marchand ennemis (proches) :
Avant la rencontre :
bateau pirate en (35, 3) battant pavillon pirate, intact
navire marchand en (35, 2) battant pavillon français, intact
Distance : 1
A l'abordage !
SOS je coule !
Apres la rencontre :
bateau pirate en (35, 3) battant pavillon pirate, intact
navire marchand en (35, 2) battant pavillon français, coulé

Deux bateaux pirates ennemis intacts (proches) :
Avant la rencontre :
bateau pirate en (35, 3) battant pavillon pirate, intact
bateau pirate en (33, 8) battant pavillon autrichien, intact
Distance : 5.38516
A l'abordage !
Non mais, ils nous attaquent ! On riposte !!
A l'abordage !
Apres la rencontre :
bateau pirate en (35, 3) battant pavillon pirate, ayant subi des dommages
bateau pirate en (33, 8) battant pavillon autrichien, ayant subi des dommages

Bateaux pirates avec dommages, ennemis :
Avant la rencontre :
bateau pirate en (35, 3) battant pavillon pirate, ayant subi des dommages
bateau pirate en (33, 8) battant pavillon autrichien, ayant subi des dommages
Distance : 5.38516
A l'abordage !
Apres la rencontre :
```

```
bateau pirate en (35, 3) battant pavillon pirate, ayant subi des dommages
bateau pirate en (33, 8) battant pavillon autrichien, coulé

Bateaux marchands ennemis :
Avant la rencontre :
navire marchand en (21, 7) battant pavillon français, intact
navire marchand en (27, 2) battant pavillon autrichien, intact
Distance : 7.81025
On vous aura ! (insultes)
Même pas peur !
Apres la rencontre :
navire marchand en (21, 7) battant pavillon français, intact
navire marchand en (27, 2) battant pavillon autrichien, intact

Pirate vs Felon :
Avant la rencontre :
bateau pirate en (33, 8) battant pavillon autrichien, intact
navire félon en (32, 10) battant pavillon français, intact
Distance : 2.23607
A l'abordage !
Même pas peur !
Apres la rencontre :
bateau pirate en (33, 8) battant pavillon autrichien, intact
navire félon en (32, 10) battant pavillon français, ayant subi des dommages

Felon vs Pirate :
Avant la rencontre :
navire félon en (32, 10) battant pavillon français, ayant subi des dommages
bateau pirate en (33, 8) battant pavillon autrichien, intact
Distance : 2.23607
A l'abordage !
Non mais, ils nous attaquent ! On riposte !!
A l'abordage !
Apres la rencontre :
navire félon en (32, 10) battant pavillon français, coulé
bateau pirate en (33, 8) battant pavillon autrichien, ayant subi des dommages
```