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

Fifth and last assignment
Final overview

J.-C. Chappelier & J. Sam

1 Exercise 1 — Cooking

The purpose of this exercise is to develop a few functionalities to manage cooking recipes.

1.1 Description

Download the source code available at the course webpage and complete it according to the instructions below.

WARNING: you should modify neither the beginning nor the end of the provided file. It's thus mandatory to proceed as follows:

1. save the downloaded file as `restaurant.cc` or `restaurant.cpp`;
2. write your code between these two provided comments:

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

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

3. save and test your program to be sure that it works properly; try for instance the values used in the example given below;

4. upload the modified file (still named `restaurant.cc` or `restaurant.cpp`) in "My submission" then "Create submission".

The provided code creates a recipe based on a selection of products, displays it, extracts the total quantity of a given product from the recipe and adapts the recipe to a different quantity (of the same product).

A possible execution example is provided further below.

1.2 Classes to produce

A *recipe* is composed of a list of *ingredients*. It has a name and is also characterized by a replication factor, *i.e.* the number of times the recipe will be prepared (a `double`, one can prepare one and a half times the recipe). We will call this attribute `nbFois_` in the reminder.

An ingredient is composed of:

- a *product*;
- a quantity (of the given product, a `double`).

A product is characterized by:

- its name (a string, `"beurre"` (means `"butter"`) for example);
- the unit of measurement typically used for this kind of product (also a string, `"grammes"` (means `"gramms"`) for example).

It can be *cooked*, *i.e.* prepared based on a recipe. A cooked product shall therefore also be characterized by its recipe.

You shall implement the classes: `Produit` (means "`Product`"), `Ingredient`, `Recette` (means "`Recipe`") and `ProduitCuisine` (means "`CookedProduct`") corresponding to the above description.

All lists shall be modeled using the `vector` type. New elements must always be added **to the end of the list**.

You will add no constructor (or assignment operator) but the ones suggested in the instructions.

The `Produit` class The public methods of the `Produit` class will start off being:

- a constructor that initializes the name and the unit of measurement associated to the product to values taken as parameters (in that order); by default, the unit is the empty string;
- the getters `getNom()` (means `getName()`) and `getUnite()` (means `getUnit()`);
- a method `toString()` simply returning the product name, without any modification.

The `Ingredient` class The public methods of the `Ingredient` class will be:

- a constructor that initializes the product and its quantity to values taken as parameters in that order (the product will be initialized with the reference of the argument product, without copying);
- the getters `getProduit()` (means `getProduct()`) and `getQuantite()` (means `getQuantity()`);

  NOTE: `getProduit` must return a constant reference to the `Produit`.
- a method `descriptionAdaptee()` (means "description of the ingredient adapted to the quantity") that produces a string representation of this ingredient ***strictly*** respecting the following format:
  `"<quantity> <unit> de <product_representation>"`
  where `<product_representation>` is the string representing the product. The quantities of the product must be adapted to the quantity of the `Ingredient`. You can use the`toString()` method of the class `Product` to produce this description. `<unit>` is the unit of measurement associated to the product.

  NOTE : Implementing `descriptionAdaptee()` requires using the method `adapter` (means "adapt") as described below for the products.

The `Recette` class The public methods of the `Recette` class will be:

- a constructor that initializes the name and the number of times this recipe is replicated to values taken as parameters (in that order); by default, the replication factor will be of `1.`;

- a method `void ajouter(const Produit& p, double quantite)` (means `void add(const Product& p, double quantity)`) that constructs an ingredient based on the given parameters and adds it to the ingredient list of the recipe. The quantity of the ingredient will b `nbFois_` multiplied by the quantity taken as parameter;

- a method `Recette adapter(double n)` (means `adapt`) that returns a new recipe corresponding to the current recipe replicated `n` times (its `nbFois_` data member has the value of the current recipe's `nbFois_` multiplied by `n`).

  Note here that the ingredients of the current recipe have already had their quantity multiplied by `nbFois_` (see previous method); you must therefore take this extra factor into account when computing the ingredient quantities of the new recipe;

- a method `toString()` that produces a string representation of this recipe ***strictly*** respecting the following format:

  ```
  Recette "<name>" x <nb_times>:
  1. <ingredient_1>
  2. <ingredient_2>
  ...
  ```

  where `<nb_times>` is the value of the `nbFois_` data member and `<ingredient_i>` is the string representing the i-th ingredient of the recipe[1]. See the execution example below for more details.

  Warning : the last ingredient shall **not** be followed by an end-of-line character! We suppose that all recipes have at least one ingredient.

The `ProduitCuisine` class A `ProduitCuisine` is a kind of `Produit`.

The public methods of the class `ProduitCuisine` will be:

- a constructor compatible with the provided `main()` method that initializes the name of the cooked product to a value taken as parameter; the unit

[1]see the `Ingredient` class and its `toString()` method

of measurement of cooked products will allways be `"portion(s)"`; the name of the recipe will be the same as the name of the product;

- a method `void ajouterARecette(const Produit& produit, double quantite)` (means `addToRecipe(const Product& product, double quantity))` that adds an ingredient to the product's recipe (you shall use the method `ajouter` from the `Recette` class);

- a method `const ProduitCuisine* adapter(double n)` that returns a pointer to a new cooked product corresponding to the current product whose recipe has been adapted `n` times;

- a method `toString()` that produces a representation of the `ProduitCuisine` ***strictly*** respecting the following format:

  ```
  <product>
  <recipe>
  ```

  where `<product>` corresponds to the traditional representation of a product[2] and `<recipe>` corresponds to the representation of this product's recipe[3]. See the execution example below for more details.

  Warning: the cooked product's representation must **not** be followed by an end-of-line character!

The product of an ingredient can now either be a (basic) product or a cooked product. The method `toString()` of the ingredient shall use for the `<product_representation>` that of the associated product adapted to the ingredient's quantity.

The methods `adapter, toString` must therefore be *polymorphic* for products.

Revisit the `Produit, ProduitCuine` class to add/change the required `adapter, toString` methods. Since there is nothing to adapt for a basic product (no associated recipe for which the quantities would need adapting); you shall simply return the current object (`this`).

The return type of `Produit::adapter()` must be `const Produit*`

[2]see the `Produit` class and its `toString()` method
[3]see the `Recette` class and its `toString()` method

NOTE: The `adapter` methods will only be used in your program by `Ingredient::descriptionAdaptee()`). You will assume however that the `adapter` methods will be used in more general contexts (they will be tested individually by the automatic grader).

The `quantiteTotale` method Finally, we wish to provide functionality for computing the total necessary quantity of a given product (identified by its name) in a recipe. The public method
`double quantiteTotale(const string& nomProduit)`
(means `double totalQuantity(const string& productName)`) of the class `Recette` will search for the given product in its ingredient list (which can also consist in cooked products that can themselves contain the sought product in their recipe). In order to enable this feature, implement the methods
`double quantiteTotale(const string& nomProduit)` in:

- the `Recette` class: it will return the sum of the total quantities of the given product taken over all the recipe's ingredients;
- the `Produit` class: it will return `1.` if this product is named `nomProduit`, and `0.` otherwise;
- the `ProduitCuisine` class: it will return `1.` if the cooked product is named `nomProduit` and otherwise, the total quantity of the given product in the recipe of the specific cooked product (`ProduitCuisine`). This method should overrides the one defined in `Produit`;
- the `Ingredient` class: it will return the ingredient's quantity multiplied by the total quantity obtained in the ingredient's product.

You are therefore asked to code the hierarchy of classes according to this description. Avoid code duplication, pay attention to the access modifiers of the attributes and methods and name the classes as suggested in the statement.

1.3 Execution example

```
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 Exercise 2 — Battleship

We are interested in this exercise to model in a very basic way a battleship game with *ships*. These can be *pirate ships* or *merchant ships*, or *traitor ships* which are both *pirate* and *merchant*.

2.1 Description

Download the source code available at the course webpage and complete it according to the instructions below.

WARNING: you should modify neither the beginning nor the end of the provided file. It's thus mandatory to proceed as follows:

1. save the downloaded file as `bateaux.cc` or `bateaux.cpp`;
2. write your code between these two provided comments:

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

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

3. save and test your program to be sure that it works properly; try for instance the values used in the example given below;
4. upload the modified file (still named `bateaux.cc` or `bateaux.cpp`) in "My submission" then "Create submission".

The provided code contains some utilities. It also contains a `main` which comprises three parts respectively described in the following sections.

Among the provided elements you will find :

- an enumerated type representing the flags (a flag is named "pavillon" in French) that can be flown by the ships ;
- an enumerated type representing the state of a ship ("State" translates to "Etat") : intact, damaged ("endommagé" in French) or sunk ("coulé" in French) ;
- a utility function called `sq(int)` returning the square on the integer passed as parameter :
- a class `Coordonnees` (means "Coordinates" in French) which models a pair of integer coordinates (typically in a imaginary grid);
- a `main` function with which you own code must *strictly* comply ; use it as a reference when coding you own classes and methods.

The code you will have to write is described below. It must be well encapsulated and avoid any code duplication.

2.2 Completing the provided utilities

In the authorized segment, after the commentary

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

and after the bracket closing the `Navire` class ("Navire" means "Ship" in French), start by coding the following elements :

- the operator += for the class`Coordonnees` which will add coordinates (useful for moving ships) ; let `(x,y)` be the pair of coordinates passed as parameter to this operator; += will add `x` and `y` respectively to the `x` and `y` coordinate of this `Coordonnees` ;

- the definition of a function named `distance` returning (as a `double`) the distance between two `Coordonnees` ; the distance between two `Coordonnees` (x_1, y_1) and (x_2, y_2) is computed according to the following formula :

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

  an overload of the display operator << for the class `Coordonnees` ; the display format is : « `(<x>, <y>)` », where `<x>` is the value of the first coordinate and `<y>` the value of the second one ; for example :
  `(1, 2)`

- an overload of the display operator << for the type `Pavillon` ; the displayed text is :

  - `"pirate"` for the value `JollyRogers` ;
  - `"français"` for the value `CompagnieDuSenegal` ;
  - `"autrichien"` for the value `CompagnieDOstende` (for sure Ostende is in Belgium now, but at that time it was under Austrian occupation !) ;
  - and `"pavillon inconnu"` for all other values ;

- an overload of the display operator << for the type `Etat` ; the displayed text is :

  - `"intact"` for the value `Intact` ;
  - `"ayant subi des dommages"` for the value `Endommage` ;
  - `"coulé"` for the value `Coule` ;
  - and `"état inconnu"` (means "unknown state") for all other values.

2.3 The `Navire` class

A `Navire` (ship) is characterized by:

- a pair of coordinates (use the provided class) allowing to retrieve its position on the grid ; name this attribute `position_` (don't forget the final underscore) ;

- a *flag* (of type `Pavillon`) named `pavillon_` indicating to which compagny the ship belongs to;

- an information named `etat_` indicating the state of the ship (use the provided type `Etat`) ;

The class `Navire` will contain:

- a constructor, compatible with the provided `main` method, initializing the coordinates and the flag of the ship using values passed as parameters; the « constructed » ship will be *intact* ;
- a method `position` returning the coordinates of the ship ;
- a method `void avancer(int de_x, int de_y)` moving the ship `de_x` units horizontally and `de_y` units vertically if it is not sunk ; `de_x` and `de_y` can be negative numbers ; use the operator `+=` to implement `avancer` ;
- a method `void renflouer()` ("renflouer" means "refloat" in French) resetting the ship to the `Intact` state ;
- a method `afficher` (means "display") displaying the ship as shown in the execution provided below, typically :

  ```
  <nom générique> en (<x>, <y>) battant pavillon <pavillon>,
  <etat>
  ```
  (on one single line) where `<nom générique>` is the type of the ship :"bateau pirate", "navire marchand" (means "merchant ship") or "navire félon" (means "traitor ship"), `<pavillon>`, is the display of the flag, `<x>` and `<y>`, are the coordinates and `<etat>` is the display of the state ;

You will also provide a function `distance` returning the distance between two `Navire` and an overload of the `<<` operator for the ships (producing the same output as `afficher`).

Also add to the class `Navire` a constant class data member named `rayon_rencontre` (means "meeting radius") having 10 as value.

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

A ship can be a pirate ship (`Pirate` class), a merchant ship (`Marchand` class) or a traitor ship (`Felon`, which is both merchant and pirate). These classes will have constructors compatible with the provided `main`.

The traitor ship must have a *single* state, a *single* position and a *single* flag, while avoiding as much code duplication as possible (more precisions are given below when explaining battleships). The `Pirate` and `Marchand` classes can be tested using the segment of the main program lying just after `Test de la partie 1` (see the execution example given below).

Encounters In order to simulate the battleship game, we adopt the following criteria: If two ships (that are not sunk) have different flags and are such that the distance which separates them is inferior to `Navire::rayon_rencontre`, they confront each other, if not, nothing happens. You are asked to provide your classes with

- a method `attaque()` (means "attack") taking another ship as parameter, and which describes how a ship attacks another ; a method `replique()` (means "respond") taking another ship as parameter, and which describes how a ship responds when it is attacked by another ship ;
- a method `est_touche` (means "`is_hit`") with no parameter, which describes what happens when a ship is hit ;
- and a method `rencontrer()` (means "encounter") taking another ship as parameter and handling the encouter with that ship; this method :
  - tests whether the conditions for a confrontation to occur are met ;
  - if yes, the `attaque` method will be called then the `replique` method (which will received the other ship as parameter).

The methods `attaque()`, `replique` and `est_touche` cannot be defined concretly in the class `Navire` ; the definition of these methods must however be imposed in all the specialized versions of ships.

The confrontation rules for specialized ships as as follows :

1. if a pirate ship `b1` attacks another ship `b2`, it "shouts" (= display) « `A l'abordage !` » (means "All aboard !") ; `b2` is then hit (method `est_touche`) ;
2. when a pirate ship `b1` responds to another ship's (`b2`) attack, it does nothing if it is marked as sunk; otherwise it displays « `Non mais, ils nous attaquent ! On riposte !!` » (means "we are attacked, let's fight back") and then attacks `b2` ;
3. if a pirate ship is hit, its state is "decreased by one level" : it moves from intact to damaged and from damaged to sunk ;

4. if a merchant ship `b1` attacks another ship `b2`, it " calls it names " : « `On vous aura ! (insultes)` » and nothing else happens ;
5. if a merchant ship `b1` responds to another ship `b2`, it displays " `SOS je coule !` " (means "SOS I'm sinking") if it is marked as sunk and " `Même pas peur !` " (means "Not even afraid!") otherwise ;
6. if a merchant ship is hit, it sinks ;
7. a traitor ships
   - attacks and is hit like a pirate one ;
   - respond like a merchant one ;

Different encounter scenarios can be tested using the segment of the main program lying just after `Test de la partie 2` (see the execution example given below).

2.5 Execution example

```
===== 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
```