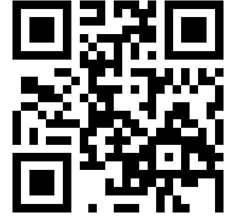
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE
EIDGENÖSSISCHE TECHNISCHE HOCHSCHULE – LAUSANNE
POLITECNICO FEDERALE – LOSANNA
SWISS FEDERAL INSTITUTE OF TECHNOLOGY – LAUSANNE

Faculté Informatique et Communications
Cours de Programmation Orientée Objet, section PH
Chappelier J.-C.

NOM : Hanon Ymous (000000)
Place : 0

#0000



Programmation Orientée Objet (SPH) : MIDTERM

17 avril 2025

INSTRUCTIONS (à lire attentivement)

IMPORTANT ! Veuillez suivre les instructions suivantes à la lettre sous peine de voir votre série annulée dans le cas contraire.

1. Vous disposez de 1h45 pour faire cette série notée (9h15 - 11h00).
2. Vous devez **écrire à l'encre noire ou bleu foncée**, pas de crayon ni d'autre couleur.

   N'utilisez **pas non plus de stylo effaçable** (perte de l'information à la chaleur).
3. Vous avez droit à toute documentation papier.

   En revanche, vous ne pouvez pas utiliser d'ordinateur personnel, ni de téléphone portable, ni aucun autre matériel électronique.
4. Répondez aux questions directement sur la donnée. Ne joignez aucune feuille supplémentaire ; **seul ce document sera corrigé**.
5. Lisez attentivement et *complètement* les questions de façon à ne faire que ce qui vous est demandé. Si l'énoncé ne vous paraît pas clair, demandez des précisions à l'un(e) des assistant(e)s.
6. Cette série notée ne comporte qu'un seul exercice (en 9 questions ; total : 98 points).

Exercice 1 – Jeu de plateau

Cadre général

On s'intéresse ici à simuler un jeu de plateau[1] constitué de plusieurs sortes de tuiles[2] ayant différents coûts et différentes façons de rapporter des points aux joueurs.

Nous fournissons plus de détails plus loin, mais globalement nous allons vous demander d'écrire diverses *portions* d'un programme C++ permettant de jouer à ce jeu, en utilisant une approche « orientée objets » avec le moins de duplication de code possible et sans aucune fuite de mémoire.

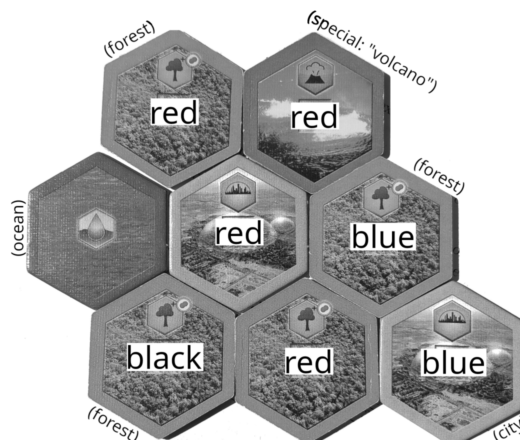
Tous les éléments du programme (joueurs, tuiles, jeu, ..) doivent pouvoir être affichés de façon spécifique au moyen de l'opérateur <<. Pour cela, vous avez déjà à disposition la classe suivante :

```
class Printable {
public:
  virtual void display(ostream& out) const = 0;
  virtual ~Printable() = default;
  // rule of 3 (or 5)
  // ...
};

ostream& operator<<(ostream& out, const Printable& something)
{
  something.display(out);
  return out;
}
```

Pour donner une idée générale, nous fournissons sur la page ci-contre un exemple de `main()` *possible* et ci-dessous le résultat attendu correspondant. Cet exemple de `main()` peut être adapté à votre conception si nécessaire. Justifiez/expliquez alors vos changements (lorsque vous les définirez).

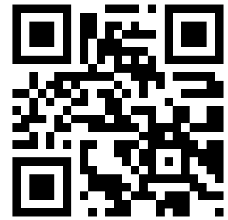
Sortie attendue pour le `main()` fourni sur la page ci-contre :

```
Joueur 1 (ready) :
red (money=50, points=0)

Mise en place du jeu :
red a payé 25, reste : 25
blue a payé 25, reste : 25
red a payé 7, reste : 18
red a payé 7, reste : 11
blue a payé 7, reste : 18
black a payé 7, reste : 43
red a payé 5, reste : 6

Ajout d'une tuile :
ville de blue (money=18, points=0)

Scores :
red (money=6, points=9)
blue (money=18, points=7)
black (money=43, points=1)
```

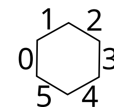


1. "*board game*" in English. 2. "*tiles*" in English.

Ne pas écrire dans cette zone.



```
int main()
{
  // players
  array<Player, 3> players({ Player("red"), Player("blue"), Player("black") });

  cout << "Joueur 1 (ready) : " << endl;
  cout << players[0] << endl << endl;

  // game tiles at that moment (snapshot)
  cout << "Mise en place du jeu :" << endl;
  Ocean   eleven;
  City    osgiliath(players[0]);    City   minastirith(players[1]);
  Forest  gump      (players[0]);    Forest broceliande(players[0]);
  Forest  fangorn   (players[1]);    Forest forbidden  (players[2]);
  Special amonamarth("volcano", players[0], 5, 0);

  // board situation at that moment
  Board board({
        &gump, &amonamarth,
    &eleven, &osgiliath, &fangorn,
      &forbidden, &broceliande
    });

  // display and add one tile
  cout << endl << "Ajout d'une tuile :" << endl;
  cout << minastirith << endl << endl;
  board.add(&minastirith);

  // create neighborhood links
  board.link(0, 1, 3); // tile 1, tile 2, side of tile 1
  board.link(0, 3, 4);
  board.link(0, 2, 5);

  board.link(1, 4, 4);
  // ... etc.

  // compute score
  board.score();

  // show players
  cout << "Scores :" << endl;
  for (auto const& player : players) cout << player << endl;

  return 0;
}
```



Un exemple *possible* de numérotation des positions des voisins.

Ne pas écrire dans cette zone.

suite au dos ☞

Question 1.1 – Les joueurs [sur 14 points]

Commencez par créer la classe `Player` permettant la représentation des joueurs.

Chaque joueur a :
- un nom, qui ne change pas (`"red"`, `"blue"`, `"black"` dans l'exemple fourni) ;
- une certaine quantité d'argent, de type `Currency` que l'on supposera fourni ;
- un certain nombre de points, de type `Points` que l'on supposera également fourni.

Lors de l'initialisation de tout joueur, on doit fournir son nom ; si l'on ne fournit pas sa quantité d'argent, elle sera initialisée à 50 ; et le nombre de points est de toute façon initialisé à 0.

Au moyen d'une méthode `pay()`, chaque joueur devra pouvoir dépenser/perdre [3] un certain montant [4] de son argent :
si le montant à dépenser [5] est plus grand que la somme disponible, on lancera une exception (de votre choix) ;
par ailleurs, pour faciliter le suivi du déroulement de la partie, à chaque fois qu'un joueur dépense de l'argent, on affichera un message au format (voir aussi des exemples dans l'exemple fourni départ) :
`<name> a payé <amount>, reste <total money>`

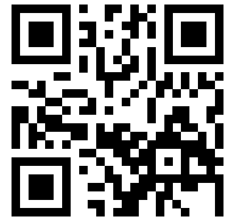
On doit par ailleurs pouvoir ajouter des points au joueur.

Et n'oubliez pas que les joueurs doivent pouvoir être affichés en utilisant l'opérateur usuel (`<<`), au format donné dans l'exemple de départ.

Réponse :

Ne pas écrire dans cette zone.

3. "*pay*" in English. 4. "*amount*" in English. 5. "*the amount to pay*" in English.



Ne pas écrire dans cette zone.

Question 1.2 – Les tuiles [sur 71 points]

Ce jeu est composé de différentes tuiles[6] (océans, forêts, villes et « tuiles spéciales ») qui peuvent chacune avoir jusqu'à 6 tuiles voisines (tuiles hexagonales, cf illustration en début de donnée).

Question 1.2.1 – Les tuiles génériques [sur 17 points]

Chaque tuile devra donc avoir un moyen d'accéder à ses six voisins, ainsi qu'une méthode `link()` permettant de connecter un nouveau voisin (voir l'exemple de `main()` fourni au début).

Au départ, toute tuile n'a aucun voisin.

Par ailleurs, chaque tuile aura aussi :

— une méthode `score()` dont le comportement sera spécifique à chaque type de tuiles et que l'on ne sait pas spécifier pour le moment ; cette méthode ne prend pas d'argument, ne modifie pas la tuile et ne retourne aucun argument (il sera expliqué plus tard comment cette méthode fonctionne) ;

— une méthode `is_owner()` qui prend un joueur en argument et retourne vrai ou faux suivant que ce joueur est propriétaire de la tuile ou non ;
par défaut, pour une tuile quelconque, cette fonction retourne simplement `false` (mais ce comportement pourra être changé pour des tuiles particulières) ;

— une méthode `neighbor_points()` qui prend un joueur en argument et retourne des `Points` ; elle ne modifie pas la tuile en question ; par défaut, pour une tuile quelconque, cette fonction retourne simplement 0 (mais ce comportement pourra être changé pour des tuiles particulières).

Définissez ici la classe `Tile` pour représenter une tuile quelconque.
Pour la méthode `link()` ne mettez ici *que son prototype* ; sa définition fait l'objet de la sous-question suivante (page ci-contre).

Et n'oubliez pas que les tuiles doivent pouvoir être affichées en utilisant l'opérateur usuel (`<<`) ; même si on ne sait pas définir cet affichage pour une tuile quelconque.

Réponse : (vous pouvez répondre en deux colonnes si nécessaire.)

6. "*tiles*" in English.

Ne pas écrire dans cette zone.

Question 1.2.2 – La connexion des tuiles [sur 10 points]

On s'intéresse ici à définir la méthode `link()` permettant de rendre deux tuiles voisines. Cette méthode reçoit une position et « une tuile » (nous mettons ici des guillemets ; libre à vous d'interpréter ce terme comme cela convient au mieux à votre conception ; il y a aussi un exemple dans le `main()`, que vous êtes libre d'adapter).

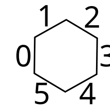
Si la position reçue est supérieure ou égale à 6, lancez une exception (de votre choix).
Si la tuile courante (`*this`) a déjà un voisin à la position reçue en argument, lancez une *autre* exception.
Et sinon, affectez la tuile reçue à la position reçue (dans les voisins de la tuile courante).

Puis faites *exactement* de même dans l'autre sens : ajoutez la tuile courante comme voisine de la tuile reçue, à la position opposée à celle reçue.

Par exemple, si l'on demande d'ajouter la tuile B à la position 3 de la tuile A, on ajoutera *aussi* la tuile A comme voisine à la position 0 de la tuile B (si possible ; sinon, on lancera les *mêmes* exceptions).
Vous supposerez pour cela qu'il existe une fonction

```
size_t opposite(size_t position);
```

qui retourne la position opposée (par exemple qui retourne 3 si elle reçoit 0 et qui retourne 0 si elle reçoit 3).



Un exemple *possible* de numérotation des positions des voisins.
En fait, peu importe : utilisez simplement la fonction `opposite()`.

Réponse :

suite au dos ☞

Ne pas écrire dans cette zone.

Question 1.2.3 – Les océans [sur 4 points]

Les tuiles océans sont les plus simples des tuiles. Elle affichent simplement « océan ». Et au niveau du score, elle ne font rien du tout (mais on peut créer des instances de tuiles océans ; voir le `main()` fourni au début).

Définissez ici votre implémentation des tuiles océans.

Réponse :

Question 1.2.4 – Tuiles possédées par des joueurs [sur 12 points]

Mises à part les tuiles océans, toutes les autres tuiles (forêts, villes et « spéciales ») sont possédées par un joueur lorsqu'elles sont en jeu.

Il est donc nécessaire qu'elles possèdent une référence (non constante) vers un joueur, lequel sera obligatoirement fourni lors de leur initialisation.

Par ailleurs, toutes ces tuiles ont un coût que les joueurs doivent payer à leur construction. Ce coût, dont la valeur par défaut est 0, sera donc passé lors de l'initialisation et de suite prélevé au joueur propriétaire (au moyen de sa méthode `pay()`).

De plus, la méthode `is_owner()` de ces tuiles doit se comporter correctement : répondre « vrai » si son argument correspond au propriétaire de la tuile et « faux » sinon.

Définissez ici une classe `Property` pour représenter de telles tuiles.

Peut-on créer des instances de `Property` ?
Répondez par oui ou par non et **justifiez** *brièvement* votre réponse.

Réponse :

Ne pas écrire dans cette zone.

(suite de la réponse pour la classe `Property` :)

Question 1.2.5 – Les forêts [sur 8 points]

Les forêts sont des tuiles ayant un joueur propriétaire et dont le coût est 7.

Au niveau de l'affichage, les forêts affichent simplement : « forêt de » suivi de l'affichage de leur propriétaire.

Au niveau du `score()`, elles apportent un `gain()` de 1 point à leur propriétaire.

Au niveau de `neighbor_points()` enfin, elles retournent 1 si le joueur passé en argument est le propriétaire de la tuile et 0 sinon.

Définissez ici une classe `Forest` pour représenter de telles tuiles.

Réponse :

Ne pas écrire dans cette zone.

suite au dos ☞

Question 1.2.6 – Les tuiles spéciales [sur 10 points]

Les tuiles spéciales sont des tuiles ayant un joueur propriétaire et dont le coût doit obligatoirement être spécifié (pas de valeur par défaut). Elles possèdent de plus un nom (qui ne sera pas modifié) et un nombre de points qui doit être fourni lors de l'initialisation et qui sera utilisé dans le `score()`.

Au niveau de l'affichage, les tuiles spéciales affichent leur nom, suivi de « de », puis de l'affichage de leur propriétaire.

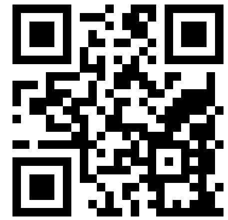
Au niveau du `score()`, elles apportent à leur propriétaire un `gain()` de leur nombre de points (celui donné lors de leur initialisation).

Pour tout le reste, elles ne font rien de plus (`is_owner()` comme toute tuile à propriétaire et `neighbor_points()` comme toute tuile générale).

Définissez ici une classe `Special` pour représenter de telles tuiles.

Réponse :

Ne pas écrire dans cette zone.



Question 1.2.7 – Les villes [sur 10 points]

Les villes sont des tuiles ayant un joueur propriétaire et dont le coût est 25.

Au niveau de l'affichage, les villes affichent simplement : « ville de » suivi de l'affichage de leur propriétaire (voir l'exemple de `main()` fourni au début).

L'autre seule spécificité des villes réside dans le calcul du `score()` : elles rapportent tout d'abord 5 points à leur propriétaire, puis, pour chaque tuile voisine, elles rapportent en plus (à leur propriétaire) le nombre de points retourné par la méthode `neighbor_points()` de la tuile voisine, appliquée au propriétaire de la tuile ville.
(Avec tout ce que nous avons défini plus haut, cela signifie qu'elles rapportent en plus 1 point pour chaque forêt adjacente qui appartient à leur propriétaire.)

Définissez ici une classe `City` pour représenter de telles tuiles.

Réponse :

Ne pas écrire dans cette zone.

suite au dos ☞

Question 1.3 – Le plateau [sur 13 points]

Pour finir, on souhaite représenter le plateau[7], qui est simplement un ensemble de tuiles.

On doit pouvoir l'initialiser soit à vide, soit en passant un ensemble initial de tuiles (comme fait par exemple dans le `main()` fourni au début).

On devra aussi pouvoir (cf le `main()` fourni au début pour chacune de ces trois méthodes) :

- ajouter une tuile au plateau (`add()`) ;
- calculer le score du jeu (`score()`) en appelant simplement la méthode `score()` de chaque tuile sur le plateau ;
- pouvoir lier (`link()`) deux tuiles du plateau ; on donnera pour cela : l'index de la première tuile, l'index de la seconde tuile et le numéro de face de la première où la seconde va la toucher ;

  en cas d'erreur, vous afficherez un message d'erreur de votre choix.

Définissez ici une classe `Board` pour représenter le plateau de jeu.

Réponse :

7. "*board*" in English.

Ne pas écrire dans cette zone.