Programmation Orientée Objet : Structures de Données Abstraites et Bibliothèques C++

Jean-Cédric Chappelier

Laboratoire d'Intelligence Artificielle
Faculté I&C

Organisation du travail (semestre)

		MOOC	déc.	cours 1 h Jeudi 8-9	exercices 2 h Jeudi 9-11
1	20.02.25		0	Intro + compil. séparée	
2	27.02.25	1. Intro POO	0	Intro POO	
3	06.03.25	2. Constructeurs/Dest	0	Constructeurs	
4	13.03.25	3. Surcharge des opé	0	Surcharge	
5	20.03.25	4. Héritage	0	Héritage	
6	27.03.25	5. Polymorphisme	0	Polymorphisme 1	
7	03.04.25		1	Polymorphisme 2 / Collections hétérogènes	
8	10.04.25	6. Héritage multiple	1	Héritage multiple	
9	17.04.25		-	-	Midtem
-	24.04.25		-	vacances Pâques	
10	01.05.25	(7. Etude de cas)	-	Templates	
11	08.05.25		-	Structure de données abstraites ; Bibliothèques	
12	15.05.25	(7. Etude de cas)	-	Bibliothèques (fin) + Révisions	
13	22.05.25		-	-	Examen
14	29.05.25		-	(Ascension)	

Objectifs de la leçon d'aujourd'hui

- Concepts fondamentaux
- Étude de cas ?
- Retour sur la série notée

Jean-Cédric Chappelier
& Jamila Sam
EPFL

Points importants (sur 2 semaines)

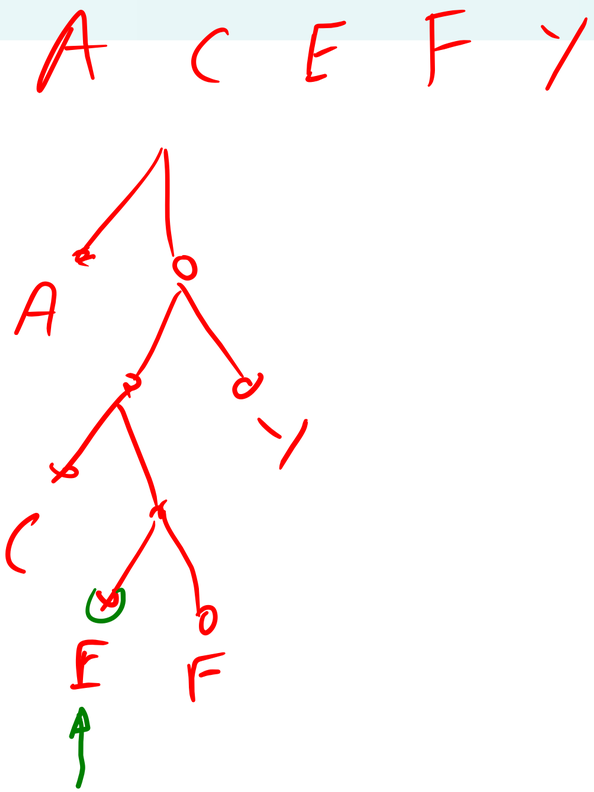
- comprendre la notion de « structure de donnée abstraite »
- savoir ce qu'est une « liste chaînée » et une « pile »
- (si ce n'est pas encore le cas) notion de pré-déclaration
- notion de « container » C++
- types :
  - `list` et `forward_list`
  - `stack` et `queue`
  - `set` et `unordered_set`
  - `map` et `unordered_map`
- notion d'itérateur
- savoir utiliser `erase()`, `find()` et `sort()`
- ... ...et continuer son apprentissage des bibliothèques C++

  ☞ Conseil : https://en.cppreference.com/

Jean-Cédric Chappelier
& Jamila Sam

Etude de cas ?

- reprendre un exemple du cours ?
- questions ?

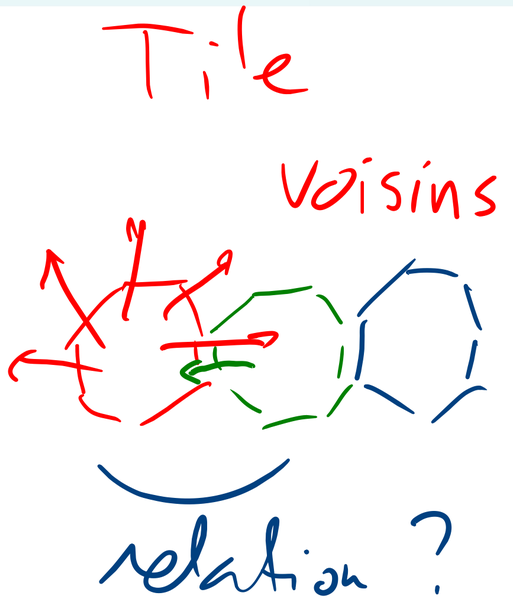



```
class IntRangeIterator {
public:
  IntRangeIterator(int index) : index_(index) {}

  bool operator==(const IntRangeIterator& x) const
  { return index_ == x.index_;  }

  bool operator!=(const IntRangeIterator& x) const
  { return not(*this == x);  }

  int operator*() const  { return index_; }
  IntRangeIterator& operator++() { ++index_; return *this; }

private:   int index_; // se souvient où l'on est
};
```

Programmati

Etude de cas ?

- reprendre un exemple du cours ?
- questions ?



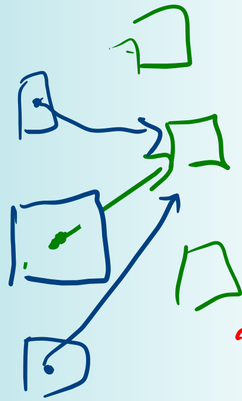
- retour sur la série notée
  - exceptions
  - voisins : quel container ? quels pointeurs ? initialisation ? destruction ?
  - `Tile::line()` pas de copié-collé... mais pas de récursion non plus !
  - propriétaires : comment le représenter ? `is_owner()` ? `==` ?
  - `City::score()` quid is le voisin n'existe pas ?
  - construction par défaut de `Board`
  - `Board::link()`

Jean-Cédric Chappelier
& Jamila Sam
EPFL

Quels pointeurs ?

POURQUOI ?

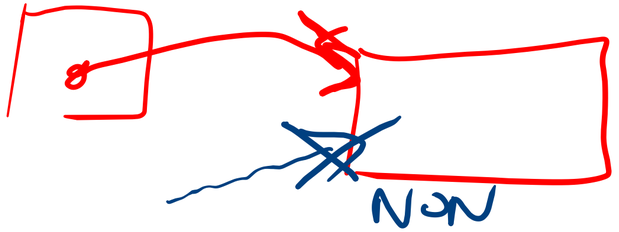
RAPPEL :



utilisation	sur des données	sur des fonctions
référence	références (ou pointeurs à la C)	nom de la fonction
généricité	pointeur à la C ou index dans un tableau	`std::function` ou pointeur à la C
allocation dynamique durée vie > portée	smart-pointers surtout `unique_ptr` (ou pointeurs à la C)	—

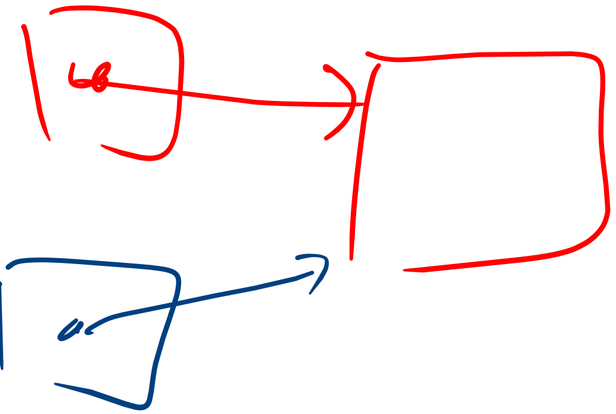
« *Utilisez des références quand vous pouvez, des pointeurs quand vous devez.* »

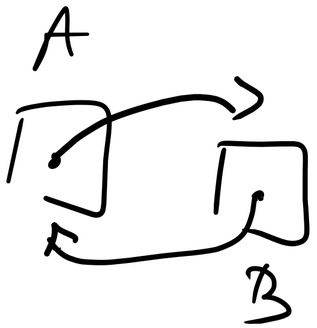
Unique :

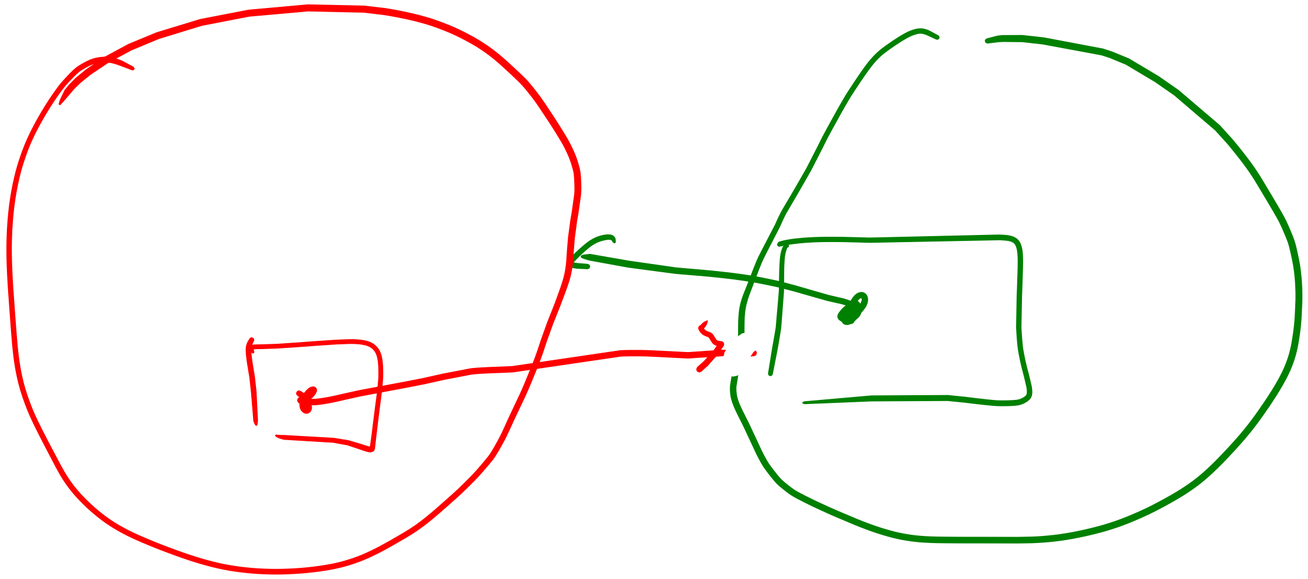


Shared

oui!







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.

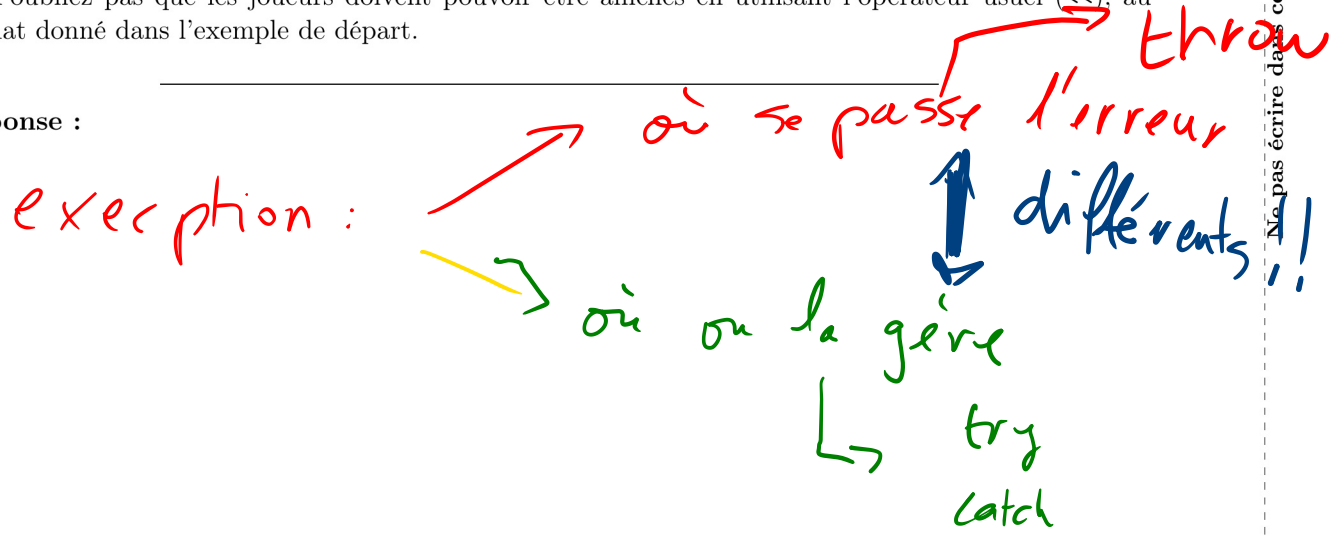
Lors de l'initialisation de tout joueur, on doit fournir son nom ; si l'on ne fournit pas sa quantité d'argent, elle sera initialisée à 50 ; et le nombre de points est de toute façon initialisé à 0.

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 :



No pas écrire dans cette zone.

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

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 tile 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 :

player1 == player2
oui MAIS
si surcharge ==

class Property
{

Player& prop;

ne pas changer
doit préexister

};

Ne pas écrire dans cette zone.