C++ : gestion mémoire

# TP C++

*Cycle de vie et gestion mémoire*

Eric Ramat
eric.ramat@univ-littoral.fr

2 novembre 2023

Durée : 6 heures

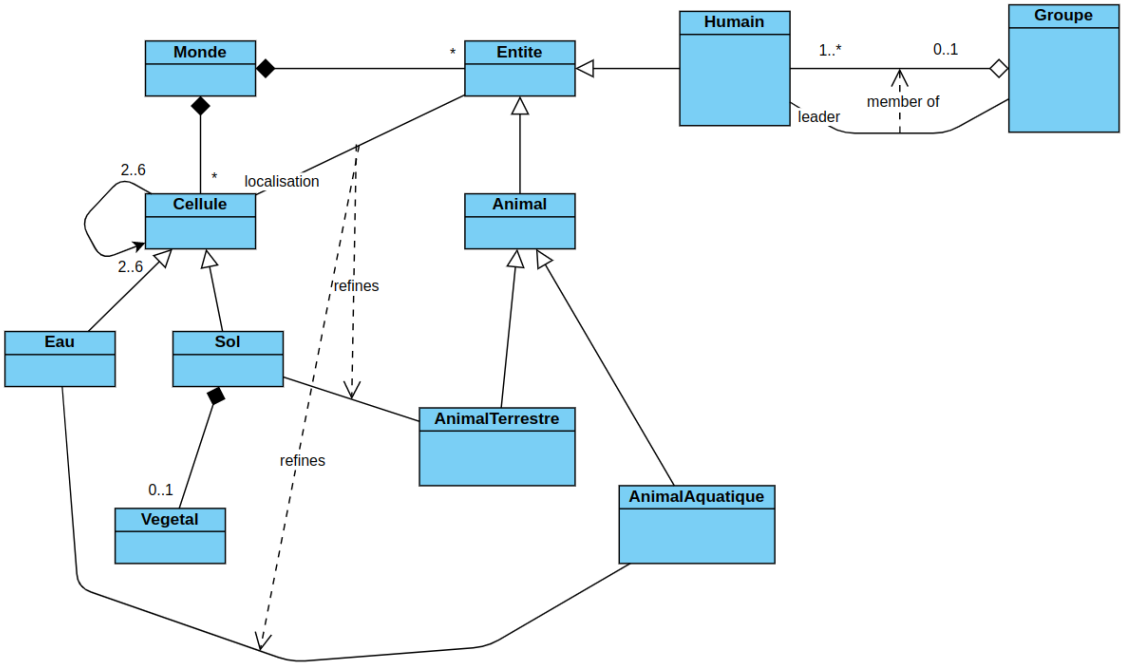
## 1 Objectifs

Ce premier TP a pour objectif de comprendre les différents mécanismes de gestion de la mémoire en C++. On va utiliser deux techniques :

- la création d'instance sous forme de variables locales ; cette dernière est créée à la déclaration, elle est initialisée via le constructeur et elle est détruite à la fin du bloc où elle est déclarée
- l'allocation dynamique ; l'objet est alors créé et initialisé lors de l'allocation mémoire et il est détruit lorsque l'on invoqué la fonction `delete`

Nous allons revoir ces deux techniques et nous allons exploiter deux autres mécanismes : les références et les pointeurs intelligents.

Afin d'illustrer les diverses techniques, on va implémenter une partie du diagramme UML de classes ci-dessous. Ce diagramme modélise les éléments d'un monde.



# 2 Travail demandé

## 2.1 Avant de commencer ...

Avant de commencer le TP, il est nécessaire de forker le dépôt `git` contenant le projet avec un début de code. Vous devez voir apparaître dans votre compte gitlab (gitlab.dpt-info.univ-littoral.fr) le projet "mini_world". Après avoir forker, à l'aide de la commande `git clone` sur votre machine, vous pouvez télécharger le code sur votre compte (en local). A ce stade, il est possible de vérifier que tout s'est bien passé en lançant la compilation à l'aide de la commande `cmake` :

```
$ mkdir build
$ cd build
$ cmake ..
$ make
$ ./src/main
```

**La réponse à chaque question fera OBLIGATOIREMENT l'objet d'un commit dont le message sera de la forme "TP1 - Question x".**

## 2.2 Rappel : compilation séparée

Afin d'optimiser la compilation, il faut créér deux fichiers par struct : un fichier d'entêtes (.hpp) et un fichier contenant le corps des fonctions (.cpp).

**Attention à bien protéger vos fichiers contre la double inclusion !**

N'oubliez pas de modifier le fichier CMakeLists.txt du répertoire src afin qu'une modification dans un fichier `.hpp` ou `.cpp` n'engendre pas la compilation de la totalité du code.

## 2.3 Un peu d'affichage

Dans le code initial, une structure est déjà présente : `World`. C'est l'implémentation de l'entité "Monde" du diagramme UML de classes. Elle représente le monde dans lequel des entités évoluent.

**Question 1.** Ajouter dans la structure `World` définie dans les fichier `world.hpp` et `world.cpp`, un constructeur et un destructeur avec un message qui s'affichera dans la console lors de leur exécution. Vous devez obtenir la sortie suivante :

```
World: constructor
World: destructor
```

**Question 2.** La structure `World` ne contient aucune donnée. Ajouter un attribut `name` de type `string` qui contiendra le nom du monde. Modifier le constructeur afin de pouvoir initialiser le nom.
Voici la nouvelle sortie console :

```
World: constructor - Eriador
World: destructor
```

## 2.4 Une nouvelle structure

Des entités évoluent dans le monde. On va donc ajouter une nouvelle structure pour représenter les entités.

**Question 3.** Ajouter la structure `Entity`. Deux nouveaux fichiers doivent être créés (`entity.hpp` et `entity.cpp`).

**Question 4.** Comme le monde, une entité possède un nom. Ajouter un attribut `name` de type `string` à la structure `Entity` et ajouter un constructeur.

**Question 5.** Une entité appartient à un monde et ne peut pas en changer. Comment peut-on représenter ce lien ? Quel attribut doit-on ajouter dans la structure `Entity` ? Comment l'initialiser ? Ajouter les instructions nécessaires dans le main pour illustrer le mécanisme. Des affichages supplémentaires seront le bienvenu.
Voici une sortie console possible :

```
World: constructor - Eriador
Entity: constructor - Frodon in Eriador
Entity: destructor
World: destructor
```

## 2.5 Changement de monde

Dans la première implémentation, une entité ne pouvait pas changer de monde. Finalement, on va l'autoriser. Plusieurs solutions sont possibles.

**Question 6.** On peut utiliser des pointeurs. Modifier l'attribut `world` de `Entity` en pointeur. Ajouter une nouvelle méthode `change_world` dans `Entity`. Modifier le reste du code pour qu'il puisse encore fonctionner.
Voici la sortie console que vous devez obtenir :

```
World: constructor - Eriador
World: constructor - Mordor
Entity: constructor - Frodon in Eriador
Entity: change world - Frodon in Mordor
Entity: destructor
World: destructor
World: destructor
```

**Question 7.** Les pointeurs sont source d'erreur. Il est donc préférable de ne pas les utiliser. Utiliser le pointeur intelligent `shared_ptr`. Pour le moment, on considère que le monde "appartient" aux entités. La sortie console doit rester identique.

**Question 8.** La déclaration des instances de `World` est un peu complexe et verbeuse. Utiliser le mot clé `auto`.

## 2.6 Les entités dans le monde

Pour le moment, les entités connaissaient le monde dans lequel elles évoluent mais le monde ne connait pas les entités. Un monde peut acceuillir plusieurs entités. Dans cette première version, on va utiliser des tableaux de taille fixe.

**Question 9.** La taille est définie sous forme d'une expression constante statique (la taille est fixée à 4) dans la structure `World` à l'aide du mot clé `constexpr`. On ne peut pas utiliser `const` car la valeur est utilisée pour définir un tableau statique.

**Question 10.** Déclarer un attribut de type tableau pour stocker les entités. On utilisera, une nouvelle fois, des pointeurs intelligents (shared_ptr). De plus, l'utilisation du type `Entity` dans le fichier `world.hpp` va provoquer une dépendance croisée. Pour la résoudre, il ne faut pas inclure le fichier `entity.hpp` mais faire une déclaration du type `Entity`.

**Question 11.** Ajouter une méthode `add_entity` dans la structure `World`. Cette méthode retourne `true` si l'ajout s'est bien passée. Si le tableau est plein alors la méthode retourne `false`. Attention, il faut ajouter quelque chose pour savoir si le tableau n'est pas plein et quelle case est libre, on ajoute alors l'entité. Pour tester, créér les nouvelles entités suivantes : Meriadoc, Peregrin et Aragorn.

Voici la sortie console que vous devez obtenir :

```
World: constructor - Eriador
World: constructor - Mordor
Entity: constructor - Frodon in Eriador
Entity: constructor - Sam in Eriador
Entity: constructor - Meriadoc in Eriador
Entity: constructor - Peregrin in Eriador
Entity: constructor - Aragorn in Eriador
World: Eriador - add entity Frodon
World: Eriador - add entity Sam
World: Eriador - add entity Meriadoc
World: Eriador - add entity Peregrin
World: Eriador - add entity Aragorn => failed
World: Mordor - add entity Frodon
Entity: change world - Frodon in Mordor
Entity: Aragorn destructor
World: Mordor destructor
```

**Question 12.** Vous remarquerez qu'il y a un souci. Certaines entités et l'un des mondes ne sont pas détruits. A votre avis, pourquoi ? Que doit-on modifier ? On doit obtenir la sortie console suivante :

```
World: constructor - Eriador
World: constructor - Mordor
Entity: constructor - Frodon in Eriador
Entity: constructor - Sam in Eriador
Entity: constructor - Meriadoc in Eriador
Entity: constructor - Peregrin in Eriador
Entity: constructor - Aragorn in Eriador
World: Eriador - add entity Frodon
```

```
World: Eriador - add entity Sam
World: Eriador - add entity Meriadoc
World: Eriador - add entity Peregrin
World: Eriador - add entity Aragorn => failed
World: Mordor - add entity Frodon
Entity: change world - Frodon in Mordor
Entity: Aragorn destructor
World: Mordor destructor
World: Eriador destructor
Entity: Peregrin destructor
Entity: Meriadoc destructor
Entity: Sam destructor
Entity: Frodon destructor
```

**Question 13.** Dans la version actuelle, une entité peut changer de monde via la méthode `change_world` mais on a oublié de changer le contenu du tableau d'entités des mondes. Ajouter une nouvelle méthode `move_entity_to` dans la structure `World`. Cette méthode admet deux paramètres : l'entité et le nouveau monde. La méthode `change_world` de la structure `Entity` devra être appelée dans cette nouvelle méthode. De plus, elle devra retourner un booléen pour indiquer si tout s'est bien passé. Par conséquent, dans le `main`, on utilisera la nouvelle méthode.

**====== FIN DE SEANCE ======**

**Question 14.** Afin de vérifier que tout est correct, ajouter dans chacune des structures une méthode `to_string` qui gère une représentation sous forme d'une chaîne de caractères les mondes et les entités.

Voici la sortie console :

```
== ENTITIES ==
Frodon [ Mordor ]
Sam [ Eriador ]
Meriadoc [ Eriador ]
Peregrin [ Eriador ]
Aragorn [ Eriador ]
== WORLDS ==
Eriador { Sam Meriadoc Peregrin }
Mordor { Frodon }
```

**Question 15.** Un monde est composé de cellules. Une cellule appartient à un seul monde. Ajouter une nouvelle structure Cell sans attribut et sans méthode, pour le moment. En revanche, définir un constructeur et un destructeur avec une sortie console pour indiquer qu'une cellule a été instanciée ou qu'une cellule a été détruite.

**Question 16.** Pour faire le lien entre le monde et ses cellules, on va développer une structure de données : *une liste simplement chaînée*. Ajouter une structure *List* et une structure Node pour représenter les chaînons de la liste. Un Node contient un lien vers le Node suivant et un lien vers le Cell qui gère. La structure List doit possèder trois méthodes : add, remove et size. La première méthode ajoute un élément Cell dans la liste en créant un nouveau Node en tête de la liste. La deuxième supprime le premier Node de

la liste. La dernière méthode retourne la taille de la liste. On comptera le nombre de nodes en parcourant la liste chaînée.

**Attention, il est impératif d'utiliser uniquement des pointeurs intelligents.**

Afin de tester le bon fonctionnement, on pourra utiliser le code ci-dessous :

```
List list;
auto c1 = std::make_unique<Cell>();
auto c2 = std::make_unique<Cell>();
auto c3 = std::make_unique<Cell>();
auto c4 = std::make_unique<Cell>();

list.add(c1);
list.add(c2);
list.add(c3);
list.add(c4);
std::cout << "size␣=␣" << list.size() << std::endl;
list.remove();
std::cout << "size␣=␣" << list.size () << std::endl;
```

**Question 17.** Utiliser la nouvelle structure `List` pour stocker les cellules d'un monde. La création des cellules se déroule dans le constructeur de la structure `World` et un nouveau paramètre `cell_number` est passé au constructeur afin de connaître le nombre de cellules à créér.
Si le nombre de cellules pour chacun des mondes est de 4, on doit obtenir une sortie console de ce type :

```
Cell: constructor
Cell: constructor
Cell: constructor
Cell: constructor
World: constructor - Eriador
Cell: constructor
Cell: constructor
Cell: constructor
Cell: constructor
World: constructor - Mordor
World: Mordor destructor
Cell: destructor
Cell: destructor
Cell: destructor
Cell: destructor
World: Eriador destructor
Cell: destructor
Cell: destructor
Cell: destructor
Cell: destructor
```

**ATTENTION, toutes les méthodes de World sont impactées par ce changement.**

**Question 18.** Pour le moment, les entités sont stockées dans un tableau statique dans World. On aimerait relier les entités aux cellules. Une entité est positionnée sur une cellule. Ajouter un attribut dans la structure Cell pour relier l'entité à une cellule. Une entité est présente dans une seule cellule.

**Question 19.** Pour identifier les cellules au sein d'un monde, ajouter un attribut id de type unsigned int. Lors de la création du monde, cet identifiant est initialisé via un paramètre du même type.

**Question 20.** Définir une nouvelle méthode *place_entity* dans World qui admet en paramètre le nom de l'entité et l'id de la cellule, est dont l'objectif est de placer l'entité dans une cellule. Il faut donc trouver l'entité dans le tableau à l'aide de son nom et la cellule dans la liste à l'aide de son id. Pour la cellule, on se propose de développer une nouvelle structure Iterator. Un Iterator est une structure qui permet de construire des itérations sur les listes. Ce type dispose d'un constructor et de trois méthodes : next, valid et cell. Le constructor admet en paramètre une référence sur la liste que l'on veut parcourir. La méthode next avance à l'élément suivant, valid retourne true si l'itérateur référence un noeud et cell retourne la cellule contenu dans le noeud.
Le code d'utilisation ressemble à :

```
List list;
Iterator it(list);

while (it.valid()) {
  auto c = it.cell();

  // ...
  it.next();
}
```

**Question 21.** Modifier la méthode to_string de World afin que l'on visualise le contenu des cellules des mondes.
Voici un exemple de code :

```
int main() {
  auto eriador = std::make_shared<World>("Eriador", 4);
  auto mordor = std::make_shared<World>("Mordor", 4);
  auto frodon = std::make_shared<Entity>("Frodon", eriador);
  auto sam = std::make_shared<Entity>("Sam", eriador);
  auto meriadoc = std::make_shared<Entity>("Meriadoc", eriador);
  auto peregrin = std::make_shared<Entity>("Peregrin", eriador);
  auto aragorn = std::make_shared<Entity>("Aragorn", mordor);

  eriador->add_entity(frodon);
  eriador->place_entity("Frodon", 0);
  eriador->add_entity(sam);
  eriador->place_entity("Sam", 1);
  eriador->add_entity(meriadoc);
  eriador->place_entity("Meriadoc", 2);
  eriador->add_entity(peregrin);
  eriador->place_entity("Peregrin", 3);
  mordor->add_entity(aragorn);
```

```
  mordor->place_entity("Aragorn", 0);

  std::cout << "==␣WORLDS␣==" << std::endl;
  std::cout << eriador->to_string() << std::endl;
  std::cout << mordor->to_string() << std::endl;

  return 0;
}
```

Une sortie possible :

```
Cell: constructor - 0
Cell: constructor - 1
Cell: constructor - 2
Cell: constructor - 3
World: constructor - Eriador
Cell: constructor - 0
Cell: constructor - 1
Cell: constructor - 2
Cell: constructor - 3
World: constructor - Mordor
Entity: constructor - Frodon in Eriador
Entity: constructor - Sam in Eriador
Entity: constructor - Meriadoc in Eriador
Entity: constructor - Peregrin in Eriador
Entity: constructor - Aragorn in Mordor
World: Eriador - add entity Frodon
World: Eriador - add entity Sam
World: Eriador - add entity Meriadoc
World: Eriador - add entity Peregrin
World: Mordor - add entity Aragorn
== WORLDS ==
Eriador { Frodon Sam Meriadoc Peregrin } { [3 / Peregrin] [2 / Meriadoc] [1 / Sam]
    [0 / Frodon] }
Mordor { Aragorn } { [3 / <empty>] [2 / <empty>] [1 / <empty>] [0 / Aragorn] }
World: Mordor destructor
Cell: destructor - 0
Cell: destructor - 1
Cell: destructor - 2
Cell: destructor - 3
Entity: Aragorn destructor
World: Eriador destructor
Cell: destructor - 0
Cell: destructor - 1
Cell: destructor - 2
Cell: destructor - 3
Entity: Peregrin destructor
Entity: Meriadoc destructor
Entity: Sam destructor
```

```
Entity: Frodon destructor
```