TP1 C++ : Première classe

# INTRODUCTION

Les objectifs de ce premier TP sont assez limités puisqu'il s'agit de la première prise en main des outils de programmation vue en cours. On a essayé ainsi d'appliquer des notions d'algorithmie, d'orienté objets et de technique spécifique tels-que l'allocation dynamique, le gcc ou le makefile. Ce TP nous a permis d'apprendre à bien structurer son code (respecter le guide de style INSA) et bien le commenté pour qu'il soit facile à lire.

# RESUME DU CAHIER DES CHARGES

L'application qu'on nous a demandée de concevoir doit manipuler une seule classe simple mais dynamique qui représente une collection d'objets quelconque. Cette classe doit ainsi contenir deux constructeurs : un pour créer une collection vierge avec une capacité fixé à la création et un deuxième pour initialiser la collection avec un tableau d'objets donné et une méthode pour afficher son contenu. Cette classe doit contenir une méthode pour ajouter et une autre pour retirer un objet donné en paramètre dans la collection, une méthode pour ajuster la taille de la collection à la taille spécifié et une méthode qui sert à réunir deux collections d'objets. Il nous faut aussi implémenter le destructeur de la classe étant donné la nature dynamique de la classe ‘collection’.

# REALISATION

Le cahier des charges a été volontairement flou et ambigu. Il a donc fallu faire plusieurs choix pour pouvoir implémenter la classe collection. Étant donné qu'il nous a été interdit d'utiliser les templates, nous avons choisi de créer une structure simple nommée ‘dog‘ qui représente les éléments contenus par notre collection.

La classe dog nous permet de mettre en évidence le fait que le type de données contenu par la classe collection pourrait être de taille quelconque. Afin d'éviter des copies de données de taille éventuellement supérieure à celle d'un pointeur, on a choisi d'utiliser un tableau de pointeurs qui pointent sur des dogs comme structure de données interne à la classe ‘collection’. La classe collection va ainsi manipuler, voir copier, le tableau de pointeur au lieu des objets de type dog directement, ce qui permet d'optimiser l’utilisation de la mémoire et le temps d’exécution de l'application.

Un autre choix qui a été pris était d'ordonner les éléments de la collection dans l'ordre d'ajout. Ce choix permet d’avoir une structure de donnée simple et légère. D’un point de vue algorithmique, le fait que les éléments ne soient pas triés selon leur valeurs augmente la complexité de certaines opérations comme le retrait, mais en réalité, le fait que l’on travaille sur un simple tableau C de pointeurs peut être très performant (mémoire alignée, mise en cache du processeur, …).

Concernant l'allocation de mémoire pour l'ajout d'objets dans la collection, nous avons préféré allouer la mémoire par paquet, c’est-à-dire, allouer plusieurs éléments vides (pointeurs vers ‘dog’ ne pointant vers aucune valeur) à la fois afin de réduire le coût des ajouts d'objets à la collection (la capacité double à chaque nouvelle allocation).

## Attributs

Les seuls attributs de la classe ‘TP1::collection’ sont :

protected:

// Tableau de pointeurs de dog

dog\*\* m\_dogs = nullptr;

// Taille du tableau m\_dogs

size\_t m\_capacity = 0;

// Taille utilisée du tableau m\_dogs

size\_t m\_size = 0;

Les attributs sont initialisés à des valeurs par défaut dans la définition de la classe (C++ 11), ce qui permet de simplifier l’implémentation des constructeurs.

## Constructeurs

Le premier constructeur demandé est un constructeur prenant en paramètre la capacité initiale de la collection, c’est-à-dire le nombre d’élément que l’on peut ajouter à la collection sans qu’il n’y ait d’allocations dynamiques. Ce constructeur est marqué avec le mots clé ‘explicit’ afin d’éviter les conversions implicites de ‘size\_t’ vers ‘TP1::collection’.

explicit collection(size\_t capacity);

Le second constructeur demandé prend en paramètre un tableau d’objets de type ‘dog’ et un entier positif indiquant la taille de ce tableau. La capacité de la collection sera alors égale à ‘size’ si le tableau est valide et 0 sinon. De plus, les objets de type ‘dog’ seront copiés avec leur constructeur de copie afin de mieux contrôler leur durée de vie au sein de la collection.

collection(const dog dogs[], size\_t size);

Nous avons en outre choisit d'empêcher le compilateur d'implémenter automatiquement le constructeur de copie et le « copy assignment operator » avec le mot clés ‘delete’ car il nous était interdis de les implémenter et la classe ‘collection’ contient un pointeur (m\_dogs) qui ne devrait pas être copié par valeur lors de la copie de l'objet de type collection :

// Empêche l'implémentation par défaut du constructeur de copie

collection(const collection&) = delete;

// Empêche l'implémentation par défaut du 'copy assignement operator'

collection& operator=(const collection&) = delete;

## Destructeur

Nous avons également ajouté un destructeur car il faut éventuellement désallouer la mémoire précédemment allouée pour les dogs et le tableau de pointeurs ‘m\_dogs’ :

virtual ~collection();

Le destructeur est marqué ‘virtual’ car il est conseillé de surcharger le destructeur si l’on hérite d’une classe (ici ‘TP1::collection’) dont le destructeur a été définit. Le destructeur appelle en fait la méthode protégée ‘void collection::disposeDogs()’ qui désalloue la mémoire si nécessaire.

## Méthode afficher

La première méthode publique demandée hormis les constructeurs et le destructeur est la méthode ‘afficher’ :

void afficher() const;

Cette méthode affiche la valeur de la collection grâce à ‘’. On considère que ce qui compose la valeur d’une collection est non seulement la liste de ‘dog‘ mais aussi la capacité de la collection. C’est pourquoi l’affichage d’une collection est de la forme : "({ <val1>, <val2>, ... }, <m\_capacity>)" <val1>, <val2>, ... les âges des chiens 1, 2, … et <m\_capacity> la capacité de la collection. Enfin, la méthode est marquée ‘’ car elle ne modifie pas l’objet courant.

## Méthode ajuster

La méthode ‘ajuster’ permet de choisir la capacité de la collection. Cependant, nous avons choisi d’interdire toute capacité inférieure au nombre de dog présent dans la collection (inférieure à m\_size) car nous pensons que le retrait d’éléments de la collection ne fait pas partie du rôle de cette méthode.

bool ajuster(size\_t capacity);

La méthode retourne un booléen indiquant si un ajustement de la capacité a bien été effectué, c’est-à-dire si la capacité donnée en paramètre est supérieure ou égale à m\_size et que cette capacitée est différente de m\_capacity. Par conséquent, si la méthode ‘ajuster’ retourne ‘true’ alors une allocation dynamique a été faite pour construire un nouveau tableau de pointeur de ‘dog‘ ayant la taille spécifiée et les pointeurs utilisés de ‘m\_dogs’ ont été copiés

## Méthode ajouter

La méthode ‘ajouter’ ajoute un ‘dog‘ donné en paramètre à la collection. Ce dog est copié pour être ajouté à la collection. Si nécessaire, cette méthode double la capacité de la collection en réallouant le tableau ‘m\_dogs’

void ajouter(const dog& dog\_to\_add);

## Méthode retirer

La méthode ‘retirer’ a deux surcharges :

La première prend en paramètre une référence constante vers un dog qui sera retiré de la collection s’il y est présent :

bool retirer(const dog& old\_dog);

La seconde prend un tableau de ‘dog‘ à retirer et la taille de ce tableau :

bool retirer(const dog dogs[], size\_t size);

Ces deux méthodes retournent un booléen indiquant si au moins un ‘dog‘ a été retiré de la collection. De plus, comme spécifié sur le cahier des charges, un appel à l’une de ces deux méthodes garanti que la capacité de la collection sera ajustée au minimum pour que la taille de la collection soit minimale, et ce, même si aucun ‘dog‘ n’a été retiré.

## Méthode réunir

La méthode ‘reunir’ copie le contenu de la collection donnée en paramètre et l’ajoute à la collection courante :

bool reunir(const collection& other);

La méthode retourne un booléen indiquant que des éléments de ‘other’ on bien été ajoutés à la collection. Si la capacité de la collection est suffisante, les éléments de ‘other’ sont ajoutés sans réallocation du tableau ‘m\_dogs’. Sinon, ‘m\_dogs’ est réalloué à une capacité de : "2 \* (m\_size + other.m\_size)".

# TESTS UNITAIRES

Afin de vérifier le bon fonctionnement de notre classe ‘collection’, nous avons établi des tests unitaires, testant chacun une méthode de la classe.

Pour nous simplifier la tâche, nous avons tout d’abord crée une fonction ‘test’ prenant en entrée un pointeur vers une fonction qui testera une méthode de la classe ‘collection’. Cette fonction ‘test’ exécute la fonction de test dans une clause try-catch et redirige la sortie de ‘std::cout’ de manière à vérifier si la sortie du test correspond à ce qui devrais être afficher si la méthode testée était correcte.

Pour créer un nouveau test unitaire, il nous suffit alors simplement de créer une fonction utilisant la méthode testée, affichant la collection et retournant ce que devrait être la sortie si la méthode était correcte. Par exemple, un test de la méthode ‘foo’ pourrais s’écrire :

const char\* test\_foo()

{

TP1::collection dogs(3);

dogs.foo(3.14);

dogs.afficher();

// Autres tests de la méthode foo ...

// On retourne ce qui devrait être affiché

return "{ 3, 1, 4 }";

}

Puis on exécute le test dans la fonction main :

test(test\_foo, "FOO !");

Les tests unitaires que nous exécutons sont : test\_lifetime, test\_afficher, test\_ajuster, test\_ajouter, test\_retirer, et test\_reunir. Ces tests sont exécutés dans cet ordre et chacun d’entre eux ne dépendent que des méthodes testées précédemment. De plus, la méthode ‘test’ numérote automatiquement les tests de façon à éviter toute erreur entre la numérotation et l’ordre d’exécution des tests.

Chaque test comprend des sous-tests correspondant à chaque cas particuliers de l’utilisation de la méthode testée. Ces sous tests ont chacun leur scope de manière à aider l’indentification des bugs.

## TEST 1 : test\_lifetime

La fonction test\_lifetime teste si les constructeurs et le destructeur de la classe fonctionnent (ne lancent pas d’exceptions). Ce test crée des instances sur la stack et la heap de la classe ‘TP1::collection’ et ne les utilise pas. Seul leur constructeur et leur destructeur devrait donc être appelés :

{ // exemple de test ayant lieu dans ‘test\_lifetime’

TP1::collection dogs(3);

}

Cependant l’optimisateur pourrait supprimer ces objets non utilisés. Nous avons donc ajouté des commandes intrinsèques du compilateur utilisé (MSVC ou gcc) autour de la fonction pour désactiver l’optimisation de ce test.

## TEST 2 : test\_afficher

Le test 2 test la méthode ‘afficher’ dans le cas d’une collection crée avec seulement une capacité initiale, et le cas où la collection est créé avec un tableau de ‘dog’.

## TEST 3 : test\_ajuster

Le test 3 test la méthode ‘ajuster’

## TEST 4: test\_ajouter

Le test 4 test la méthode ‘ajouter’ dans le cas où la collection initiale est vide (capacité nulle), dans le cas où la collection contient déjà des éléments la remplissant et dans le cas où la collection a une capacité suffisante pour ajouter l’élément sans réallocation de ‘m\_dogs’.

## TEST 5 : test\_retirer

Le test 5 test la méthode ‘retirer’

## TEST 6 : test\_reunir

Le test 6 test la méthode ‘reunir’

Chaque test assume que le précédent a réussi pour des raisons de simplicité.

# CONCLUSION : AMELIORATIONS POSSIBLES

Templates, sort (structure de données + complexes), surcharge d’opérateurs et ctrs de copie, itérateurs ?, un moyen d’accéder aux données >.<, …