TP1 C++ : Première classe

HAFEDH Jebalia
SOTIR Paul-Emmanuel

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. Nous avons essayé d'appliquer des notions d'algorithmie, d'orienté objet et de techniques spécifiques telles-que l'allocation dynamique, gcc ou makefile. Ce TP nous a permis d'apprendre à structurer notre code, respecter un guide de style et bien commenter le code.

RESUME DU CAHIER DES CHARGES

L'application que l'on nous a demandé de concevoir doit manipuler une seule classe simple mais dynamique qui représente une collection d'objets quelconque.

La classe collection doit contenir les méthodes publiques suivantes (comportements plus détaillés dans la section suivante) :

- Une méthode 'afficher' permettant d'afficher la valeur de la collection
- Une méthode 'ajuster' pour choisir la capacité de la collection
- Une méthode 'ajouter' pour ajouter un élément à la collection
- Deux méthodes 'retirer' pour retirer un élément ou un tableau d'éléments de la collection
- Une méthode 'reunir' pour ajouter les éléments d'une collection donnée en paramètre à la collection courante

Cette classe doit également contenir deux constructeurs ainsi qu'un destructeur: un constructeur pour créer une collection vierge avec une capacité donnée et un deuxième pour initialiser la collection avec un tableau d'objets donné.

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 structure '`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 supplémentaires pour le tableau `m_dogs`. Si la capacité donnée en paramètre est nulle, alors `m_dogs` sera à la valeur 'nullptr', sinon, le constructeur alloue un tableau de pointeur de dog ayant pour taille la capacité donnée. 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 ('size'). La capacité de la collection sera alors égale à 'size' si le tableau est différent de 'nullptr' 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’. Si la collection est vide au moment de l’appel (`m_dogs == nullptr`), alors la capacité allouée pour ajouter le dog donné en paramètre est de `INITIAL_ALLOCATION_SIZE` (entier constant et statique de valeur 5).

```
void ajouter(const dog& dog_to_add);
```

Méthode retirer

La méthode ‘retirer’ permet de retirer un ou des éléments de la collection. Si un dog donné en paramètre est présent plusieurs fois dans la collection, toutes les occurrences de ce dog sont retirées. Cette méthode ce base sur l’opérateur == pour comparer les dog. 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. Ces test sont définis dans le fichier "tests.h".

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.

Nous assumons pour les méthodes prenant en paramètre un tableau et la taille de ce tableau que la taille est correcte (inconvénient des tableaux C).

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' pour quatre cas différents. Un premier cas où l'on ajuste la collection à une capacité inférieure à sa taille utilisée (ne doit avoir aucun effet sur la collection et retourner '`false`'). Un second cas où l'on ajuste une collection vide de capacité 5 à une capacité nulle. Un troisième où l'on ajuste la capacité pour l'augmenter. Et un dernier où l'on ajuste la capacité pour la réduire à une valeur non nulle et correcte.

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' avec quatre sous tests. Le premier cas essaye de retirer avec un tableau invalide (`nullptr`) en paramètre. Le second retire l'ensemble des éléments de la collection avec un tableau de '`dog`'. Le troisième retire un élément présent en plusieurs exemplaires dans la collection. Le dernier sous-test retire un tableau de '`dog`' qui ne sont pas présents dans la collection.

TEST 6 : `test_reunir`

Le test 6 test la méthode 'reunir' dans 3 cas différents. Le premier sous-test réunit deux collection pleines. Le second appelle la méthode 'reunir' en donnant en paramètre une collection vide. Le dernier test réuni à une collection de capacité suffisante une collection de '`dog`' pouvant être contenue dans la première collection sans réallocation de '`m_dogs`'.

CONCLUSION ET AMELIORATIONS POSSIBLES

Nous avons conçu une classe '`TP1::collection`' répondant aux critères du cahier des charges et vérifiant tous nos tests.

Cependant, cette classe pourrait sans doute être améliorée. Par exemple nous pourrions facilement utiliser les templates pour faire des collections génériques. Nous pourrions également éviter l'utilisation de tableaux C en paramètres grâce à `std::initializer_list, std::array ou gsl::array_view` (voir https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#-bounds1-dont-use-pointer-arithmetic-use-array_view-instead). Nous aurions pus également surcharger certains opérateurs et implémenter le constructeur de copie car les objets de type '`TP1::collection`' ne sont pas copiables pour l'instant.