TP1 C++ : REALISATION

HAFEDH Jebalia
SOTIR Paul-Emmanuel Binôme B3311

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 !");
```

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 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.

Voici l'implémentation de 'test_ajuster' :

```
// Description: test de la méthode ajuster de la classe TP1::collection
//              RETOURNE: la chaine de charactère qui devrait être affichée si la méthode testée
//                        est correcte
const char* test_ajuster()
{
        TP1::dog dogsArray[3] = {
                TP1::dog(TP1::color::GREEN, 50),
                TP1::dog(TP1::color::RED, 3),
                TP1::dog(TP1::color::BLUE, 99) };

        { // Sub-test 1
                TP1::collection dogs(dogsArray, 3);
                std::cout << dogs.ajuster(2) << " ";
                dogs.afficher();
        }
        std::cout << " ";
        { // Sub-test 2
                TP1::collection dogs(5);
                std::cout << dogs.ajuster(0) << " ";
                dogs.afficher();
        }
        std::cout << " ";
        { // Sub-test 3
                TP1::collection dogs(dogsArray, 3);
                std::cout << dogs.ajuster(10) << " ";
                dogs.afficher();
        }
        std::cout << " ";
        { // Sub-test 4
                TP1::collection dogs(dogsArray, 3);
                dogs.ajuster(10);
                std::cout << dogs.ajuster(4) << " ";
                dogs.afficher();
        }

        // Return expected output
        return
                "0 ({ 50, 3, 99 }, 3) "  // Sub-test 1
                "1 ({ }, 0) "            // Sub-test 2
                "1 ({ 50, 3, 99 }, 10) " // Sub-test 3
                "1 ({ 50, 3, 99 }, 4)";  // Sub-test 4
}
```

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.

Voici l'implémentation de 'test_retirer' :

```
// Description: test de la méthode retirer de la classe TP1::collection
//              RETOURNE: la chaine de charactère qui devrait être affichée si la méthode testée
//                        est correcte
const char* test_retirer()
{
        TP1::dog dogsArray[3] = {
                TP1::dog(TP1::color::GREEN, 50),
                TP1::dog(TP1::color::RED, 3),
                TP1::dog(TP1::color::BLUE, 99) };

        { // Sub-test 1
                TP1::collection dogs(dogsArray, 3);
                std::cout << dogs.retirer(nullptr, 0) << " ";
```

```
                dogs.afficher();
        }
        std::cout << " ";
        { // Sub-test 2
                TP1::collection dogs(dogsArray, 3);
                std::cout << dogs.retirer(dogsArray, 3) << " ";
                dogs.afficher();
        }
        std::cout << " ";
        { // Sub-test 3
                TP1::collection dogs(dogsArray, 3);
                dogs.ajouter(TP1::dog(TP1::color::RED, 3));
                std::cout << dogs.retirer(dogsArray[1]) << " ";
                dogs.afficher();
        }
        std::cout << " ";
        { // Sub-test 4
                TP1::collection dogs(0);
                dogs.ajouter(TP1::dog(TP1::color::GREEN, 5));
                std::cout << dogs.retirer(dogsArray, 3) << " ";
                dogs.afficher();
        }

        // Return expected output
        return
                "0 ({ 50, 3, 99 }, 3) "  // Sub-test 1
                "1 ({ }, 0) "                    // Sub-test 2
                "1 ({ 50, 99 }, 2) "     // Sub-test 3
                "0 ({ 5 }, 1)";                  // Sub-test 4
}
```

CODE SOURCE DE LA CLASSE TP1::COLLECTION

```
/*************************************************************************
                        collection  -  A collection of dogs
                        ----------------------------------
date                 : 10/2015
copyright            : (C) 2015 by B3311
*************************************************************************/

//--------- Interface de la classe <collection.h> (fichier collection.h) ---------
#ifndef COLLECTION_H
#define COLLECTION_H

//------------------------------------------------------------ Interfaces utilisées
#include "dog.h"

namespace TP1
{
        //-------------------------------------------------------------------------
        // collection:
        // classe qui permet de manipuler une collection d'objets de type dog
        //-------------------------------------------------------------------------
        class collection
        {
                //------------------------------------------------------------------ PUBLIC
        public:
                //---------------------------------------------------- Méthodes publiques

                // Description : affiche la valeur de la collection.
                //      La valeur de la collection est affichée sous la forme : "({ <val1>, <val2>, ... },
                //      <m_capacity>)" avec <val1>, <val2>, ... les âges des chiens et <m_capacity> la
                //      capacité de la collection.
                void afficher() const;

                // Description: ajoute dog_to_add dans la collection courante
                //              new_dog: reférence constante vers un dog qui sera ajouté dans la
                //              collection
                void ajouter(const dog& dog_to_add);

                // Description: retire old_dog de la collection courante
                //              old_dog: reference vers un dog qui sera retiré de la collection
                //              RETOURNE: Un booléen indiquant si au moins un dog a bien été retiré
                //      NOTE: ajuste la capacité de la collection au minimum même si old_dog n'est pas
                //      présent dans la collection
                bool retirer(const dog& old_dog);

                // Description: retire l'ensemble des dogs du parametre 'dogs' de la collection courante
                //              dogs: tableau de dogs qui seront retirés de la collection
                //              size: taille du tableau dogs
                //              RETOURNE: Un booléen indiquant si au moins un dog a bien été retiré
                //      NOTE: ajuste la capacité de la collection au minimum même si aucun dog de dogs
                //      n'est pas présent dans la collection
                bool retirer(const dog dogs[], size_t size);

                // Description: ajuste, si possible, la capacité de la collection à la taille spécifiée
                //              capacity: nouvelle taille de mémoire allouée pour la structure de donnée
                //              interne
                //              RETOURNE: Un booléen indiquant si un ajustement de la capacité a bien été
                //              effectué
                bool ajuster(size_t capacity);

                // Description: ajoute le contenu de la collection donnée en paramètre
                //              other: reference vers la collection à réunir avec la collection courante
                //              RETOURNE: Un booléen indiquant si des dogs de 'other' on bien été ajoutés
                //              à la collection
                bool reunir(const collection& other);
```

```
        //------------------------------------------------ Surcharge d'opérateurs

        // Empêche l'implémentation par défaut du 'copy assignement operator'
        collection& operator=(const collection&) = delete;

        //-------------------------------------------- Constructeurs - destructeur

        // Empêche l'implémentation par défaut du constructeur de copie
        collection(const collection&) = delete;

        // Description: Constructeur d'une collection de taille pré-allouée donnée
        //              capacity: Taille de la nouvelle collection
        explicit collection(size_t capacity);

        // Description: Constructeur d'une collection à partir d'un ensemble de dog donné en
        //              paramètre
        //              dogs: tableau des dogs qui seront ajoutés à la collection
        //              size: taille du tableau dogs
        collection(const dog dogs[], size_t size);

        // Description: Destructeur de la colection courante
        virtual ~collection();

        //------------------------------------------------------------------ PRIVE
    protected:
        //---------------------------------------------------- Méthodes protégées

        // Description: Trouve le nombre de dog apparetenant à la fois à 'm_dogs'
        //      et au paramètre 'dogs_to_find'
        //              dogs_to_find: tableau de dogs qui seront recherchés dans la collection
        //              size: taille du tableau 'dogs_to_find'
        //              RETOURNE: le nombre de dog apparetenant à la fois à 'm_dogs' et à
        //              'dogs_to_find'
        unsigned int find_all_of(const dog dogs_to_find[], size_t size) const;

        // Description: Désalloue la mémoire allouée pour le tableau de pointeur de dog
        //      et pour les objets de type dog.
        void disposeDogs();

        //---------------------------------------------------- Attributs protégés

        // 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;

        //------------------------------------------------------ Constantes protégés

        static const size_t INITIAL_ALLOCATION_SIZE = 5;
    };
}

#endif // COLLECTION_H
```

```cpp
/*****************************************************************************
                                                collection  -  A collection of dogs
                                                ----------------------------------
date                 : 10/2015
copyright            : (C) 2015 by B3311
*****************************************************************************/

//--------- Réalisation de la classe <collection.h> (fichier collection.cpp) ---------

//------------------------------------------------------------------------------ INCLUDE

//-------------------------------------------------------------------- Include système
#include <iostream>

//------------------------------------------------------------------- Include personnel
#include "collection.h"

namespace TP1
{
	//------------------------------------------------------------------------------ PUBLIC
	void collection::afficher() const
	{
		if (m_size > 0)
		{
			std::cout << "({ ";

			for (size_t i = 0; i < m_size; ++i)
				std::cout << m_dogs[i]->age << ((i < m_size - 1) ? ", " : " }, ");

			std::cout << m_capacity << ")";
		}
		else
			std::cout << "({ }, " << m_capacity << ")";
	}

	void collection::ajouter(const dog& dog_to_add)
	{
		// Copy given dog
		dog* new_dog = new dog(dog_to_add);

		// Reallocate memory if we don't have free space anymore
		if (m_size == m_capacity)
		{
			// Double capacity at each new allocations
			ajuster(m_size > 0 ? 2 * m_size : INITIAL_ALLOCATION_SIZE);
		}

		// Append new dog value
		m_dogs[m_size++] = new_dog;
	}

	bool collection::retirer(const dog dogs_to_remove[], size_t size)
	{
		if (dogs_to_remove == nullptr || size == 0 || m_dogs == nullptr)
		{
			ajuster(m_size); // As specified, we always allocate the shorter amount of memory
					 // possible (m_size == m_capacity) after 'retirer(...)' is called
			return false;
		}

		// Find the number of dog to remove and get the index of the first one
		unsigned int removes_todo_count = find_all_of(dogs_to_remove, size);

		if (removes_todo_count == 0)
		{
			// There isn't any dog to remove
			ajuster(m_size); // As specified, we always allocate the shorter amount of memory
					 // possible (m_size == m_capacity) after 'retirer(...)' is called
			return false;
```

```
        }

        if (removes_todo_count >= m_size)
                disposeDogs(); // We remove all dogs
        else
        {
                size_t new_size = m_size - removes_todo_count;
                dog** new_dogs = new dog*[new_size];

                // Copy 'm_dogs' dogs in 'new_dogs' except those present in 'dogs_to_remove'
                size_t removes_count = 0;
                for (size_t i = 0; i < m_size; ++i)
                {
                        auto dog = m_dogs[i];

                        // Check if 'm_dogs[i]' is in 'dogs_to_remove'
                        bool is_to_copy = true;
                        for (size_t j = 0; j < size && removes_count < removes_todo_count; ++j)
                        {
                                if (dogs_to_remove[j] == *(dog))
                                {
                                        ++removes_count;
                                        is_to_copy = false;
                                }
                        }

                        if (is_to_copy)
                                new_dogs[i - removes_count] = dog;
                }

                delete[] m_dogs;
                m_dogs = new_dogs;
                m_size = new_size;
                m_capacity = new_size;
        }

        return true;
}

bool collection::retirer(const dog& old_dog)
{
        return retirer(&old_dog, 1);
}

bool collection::ajuster(size_t new_capacity)
{
        if (new_capacity < m_size || new_capacity == m_capacity)
                return false; // We forbid any dog removal during an 'collection::ajuster' call

        if (new_capacity == 0)
        {
                // We know that 'm_size == 0', 'm_capacity > 0' and 'm_dogs != nullptr'
                // We free all pre-allocated memory:
                delete[] m_dogs;
                m_dogs = nullptr;
                m_capacity = 0;
                return true;
        }

        // Allocate a new array of dog pointers
        dog** new_dogs = new dog*[new_capacity];

        // If this collection have any dog pointers, we copy them to 'new_dogs' and we delete
        // 'm_dogs'
        if (m_dogs != nullptr)
        {
                for (size_t i = 0; i < m_size; i++)
                        new_dogs[i] = m_dogs[i];
```

```
                        delete[] m_dogs;
                }

                // Assign the new array to 'm_dogs' and update 'm_capacity'
                m_dogs = new_dogs;
                m_capacity = new_capacity;

                return true;
        }

        bool collection::reunir(const collection& other)
        {
                if (other.m_size == 0 || other.m_dogs == nullptr)
                        return false; // We don't have any dogs to append

                if (m_capacity - m_size >= other.m_size)
                {
                        // We have enought free allocated space to store other.m_dogs in m_dogs
                        for (size_t i = 0; i < other.m_size; ++i)
                                m_dogs[m_size + i] = new dog(*(other.m_dogs[i]));

                        m_size += other.m_size;
                }
                else
                {
                        // Allocate a new array of dog pointers
                        dog** new_dogs = new dog*[2 * (m_size + other.m_size)];

                        // Copy other.m_dogs to new_dogs
                        for (size_t i = 0; i < other.m_size; i++)
                                new_dogs[i + m_size] = new dog(*(other.m_dogs[i]));

                        if (m_dogs != nullptr)
                        {
                                // Copy m_dogs pointers to new_dogs
                                for (size_t i = 0; i < m_size; ++i)
                                        new_dogs[i] = m_dogs[i];

                                delete[] m_dogs;
                        }

                        // Assign the new array to 'm_dogs' and update 'm_capacity' and 'm_size'
                        m_dogs = new_dogs;
                        m_size += other.m_size;
                        m_capacity = 2 * m_size;
                }

                return true;
        }

        //------------------------------------------------- Constructeurs - destructeur
        collection::collection(size_t capacity)
        {
#ifdef MAP
                cout << "Appel au constructeur 'collection::collection(size_t)'" << endl;
#endif

                if (capacity > 0)
                        m_dogs = new dog*[capacity];
                else
                        m_dogs = nullptr;
                m_capacity = capacity;
        }

        collection::collection(const dog dogs[], size_t size)
        {
#ifdef MAP
                cout << "Appel au constructeur 'collection::collection(const dog[], size_t)'" << endl;
#endif
```

```
                if (size > 0 && dogs != nullptr)
                {
                        m_capacity = size;
                        m_size = size;

                        // Create a new array of dog pointers
                        m_dogs = new dog*[size];

                        // Copy given dogs
                        for (size_t i = 0; i < size; ++i)
                                m_dogs[i] = new dog(dogs[i]);
                }
        }

        collection::~collection()
        {
#ifdef MAP
                cout << "Appel au destructeur de 'collection'" << endl;
#endif
                // On désalloue la mémoire allouée pour le tableau de pointeur de dog et pour les dogs
                // eux-même.
                disposeDogs();
        } //----- Fin de ~collection{file_base}

        //------------------------------------------------------------------------------ PRIVE

        //------------------------------------------------------------- Méthodes protégées
        unsigned int collection::find_all_of(const dog dogs_to_find[], size_t size) const
        {
                unsigned int matches_count = 0;

                if (size > 0 && dogs_to_find != nullptr && m_size > 0)
                {
                        for (size_t idx = m_size - 1; idx < m_size; --idx)
                        {
                                for (size_t idx2 = 0; idx2 < size; ++idx2)
                                {
                                        if (*(m_dogs[idx]) == dogs_to_find[idx2])
                                        {
                                                ++matches_count;
                                                break;
                                        }
                                }
                        }
                }

                return matches_count;
        }

        void collection::disposeDogs()
        {
                if (m_dogs != nullptr)
                {
                        for (size_t i = 0; i < m_size; ++i)
                        {
                                dog* dog_ptr = m_dogs[i];
                                if (dog_ptr != nullptr)
                                        delete dog_ptr;
                                m_dogs[i] = nullptr;
                        }
                        delete[] m_dogs;
                        m_dogs = nullptr;

                        m_size = 0;
                        m_capacity = 0;
                }
        }
}
```

MAKEFILE DU PROGRAMME

```
# Makefile:
CPP_FLAGS = -Wall -std=c++11

EXE: collection.o main.o
	g++ collection.o main.o -o EXE
main.o: main.cpp dog.h collection.h tests.h
	g++ $(CPP_FLAGS) -c main.cpp -o main.o
collection.o: collection.cpp collection.h dog.h
	g++ $(CPP_FLAGS) -c collection.cpp
```