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

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

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