Compte rendu du TP C++ 1
Spécification et conception

Marc Gagné et Selma Nemmaoui

B3309

1. La classe `CollectionEntiers`

La classe `CollectionEntiers` permet de gérer des collections d'entiers de taille quelconque fixée à la création avec réajustement automatique ou à la demande. Les valeurs sont stockées dans l'ordre d'insertion, sans recherche d'optimisation pour la recherche, l'insertion ou la suppression (le choix d'entiers comme type de valeur était arbitraire – la classe pourrait être facilement modifiée pour accommoder des réels, des chaînes, des booléens, etc.). Les doublons sont autorisés.

Définitions

Ce document utilise plusieurs termes spécifiques, définis ci-dessous, pour décrire le projet :

- Valeur : l'entier contenu dans un élément de la collection.
- Élément : l'unité de stockage de la collection ; la collection peut contenir un nombre variable d'éléments, chacun avec une valeur associée.
- Collection : ensemble ordonné d'éléments dont la taille peut varier (par opposition à un tableau, dont la taille est fixée).

Schéma

Le projet ne contient qu'une seule classe, la classe `CollectionEntiers`, résumée ci-dessous :

CollectionEntiers
#alloue : unsigned int #elements : unsigned int #valeurs : int *
+Afficher() +Ajouter(valeur : int) +Retirer(n : unsigned int, retirer : int[]) +Reunir(collection : CollectionEntiers *) +Ajuster(n : unsigned int) +CollectionEntiers(n : unsigned int) +CollectionEntiers(n : unsigned int, valeurs : int[]) #Init(n : unsigned int)

2. La structure de données

Pour manipuler ses éléments, la collection stocke chaque élément dans un tableau d'entiers, utilisant deux autres attributs pour gérer la lecture et la modification de ce tableau :

- `valeurs` : un pointeur vers un tableau d'entiers, qui contient les éléments de la collection, et est manipulé par les méthodes de la classe.
- `alloue` : la quantité de mémoire allouée à `valeurs` (entier positif). Modifié à chaque opération d'ajustement de `valeurs` pour refléter sa nouvelle taille.
- `elements` : le nombre d'éléments dans `valeurs` (entier positif). Toujours inférieur ou égal à `alloue`, puisqu'on ne peut pas avoir plus d'éléments qu'il n'y a de place pour les stocker. Modifié lors de l'ajout ou de la suppression d'éléments (notamment lors d'un ajustement rétrécissant de `valeurs`).

3. Les méthodes et tests fonctionnels de `CollectionEntiers`

Les méthodes principales de la collection sont présentées ci-dessous. Les tests se trouvent dans le fichier `main.cpp` ; l'exécution de la fonction `main` lance l'exécution de chacun des **16** tests. La sortie résultante doit manuellement être vérifiée par l'utilisateur, afin que l'affichage attendu corresponde à l'affichage obtenu.

`CollectionEntiers (unsigned int n, int valeurs[])`

`n` : le nombre de valeurs initiales

`valeurs` : les valeurs initiales

Construit une nouvelle collection avec un ensemble de `n` valeurs initiales, listées dans `valeurs`. Si `n` vaut `0`, ceci revient à appeler le premier constructeur avec la même valeur de `n`. Sinon, la méthode alloue suffisamment de mémoire pour stocker les `n` valeurs initiales puis elle les ajoute à la collection.

Test #3 (`test_3_construction_2`) : Vérifie que le constructeur 2 permet de créer une collection avec des valeurs initiales.

Test #4 (`test_4_construction_2_vide`) : Vérifie que le constructeur 2 permet de créer une collection vide.

Test #5 (`test_5_construction_2_affichage`) : Vérifie que le constructeur 2 permet de bien stocker les valeurs passées en paramètre, en vérifiant également que l'affichage s'effectue correctement.

`void Ajouter (int valeur)`

`valeur` : la valeur à ajouter à la collection

Vérifie s'il y a assez de mémoire allouée pour ajouter une valeur à la collection, augmente la taille de la collection de 1 dans le cas contraire, puis insère la valeur en fin de la collection.

Note : si vous planifiez d'insérer un grand nombre connu de valeurs, il est plus performant d'appeler `Ajuster` au préalable (avec, en paramètre, la nouvelle taille prévue), puis d'appeler `Ajouter` pour chaque valeur.

Test #6 (`test_6_ajouter_sans_reallocation`) : Vérifie que la méthode `Ajouter` ajoute bien la valeur passée en paramètre à une collection initialisée avec assez de mémoire allouée pour la stocker.

Test #7 (`test_7_ajouter_avec_reallocation`) : Vérifie que la méthode `Ajouter` ajoute bien la valeur passée en paramètre à une collection initialisée avec une quantité insuffisante de mémoire allouée pour la stocker (ré-allocation de place nécessaire).

`void Retirer (unsigned int n, int retirer[])`

`n` : le nombre de valeurs à supprimer

`retirer` : la liste des valeurs à retirer de la collection

Supprime chaque entier listé dans le tableau fourni de la collection. Si une valeur à supprimer n'est pas trouvée dans la collection, aucune action n'est prise. Les doublons dans `retirer` sont autorisés, mais superflus.

Test #8 (`test_8_retirer`) : Vérifie que la méthode `Retirer` retire toutes les valeurs passées en paramètre d'une collection non vide.

Test #9 (`test_9_retirer_vide`) : Vérifie que la méthode `Retirer` fonctionne sur une collection vide.

Test #10 (`test_10_retirer_aucun`) : Vérifie que la méthode `Retirer` fonctionne si on ne spécifie aucune valeur à retirer.

`void Reunir (CollectionEntiers * collection)`

`collection` : la collection contenant les éléments à ajouter

Vérifie s'il y a assez de mémoire allouée pour ajouter tous les éléments de `collection`, augmente la taille de cette collection dans le cas contraire, puis y ajoute tous les éléments de `collection`.

Test #14 (`test_14_reunir_sans_reallocation`) : Vérifie que la méthode `Reunir` réunit deux collections sachant que la première collection possède assez de place à allouer à la deuxième (pas d'ajustement nécessaire).

Test #15 (`test_15_reunir_avec_reallocation`) : Vérifie que la méthode `Reunir` réunit deux collections sachant que l'espace alloué à la collection qui sera modifiée sera insuffisant lors de la réunion (ajustement nécessaire).

Compte rendu du TP C++ 1
Réalisation

Marc Gagné et Selma Nemmaoui

B3309

1. Réalisation des tests fonctionnels

void Retirer (unsigned int n, int retirer[])

test_8_retirer :

```
bool test_8_retirer ()
// Algorithme : Construit une nouvelle collection avec des valeurs
// initiales, puis essaie d'en retirer certaines avant d'afficher le
// résultat.
{
    cout << "\tAffichage attendu :\t1 3 4 " << endl;
    cout << "\tAffichage obtenu :\t";
    unsigned int taille = 5;
    int valeurs[] = {1, 2, 3, 4, 5};
    unsigned int n = 3;
    int retirer[] = {5, 2, 5, 6};
    CollectionEntiers * c = new CollectionEntiers(taille, valeurs);
    c->Retirer(n, retirer);
    c->Afficher();
    delete c;
    return true;
}
```

test_9_retirer_vide :

```
bool test_9_retirer_vide ()
// Algorithme : Construit une nouvelle collection de taille nulle, puis
// essaie d'en retirer certaines valeurs avant d'afficher le résultat.
{
    cout << "\tAffichage attendu :\t " << endl;
    cout << "\tAffichage obtenu :\t";
    unsigned int taille = 0;
    unsigned int n = 3;
    int retirer[] = {5, 2, 5};
    CollectionEntiers * c = new CollectionEntiers(taille);
    c->Retirer(n, retirer);
    c->Afficher();
    delete c;
    return true;
}
```

test_10_retirer_aucun :

```
bool test_10_retirer_aucun ()
// Algorithme : Construit une nouvelle collection avec des valeurs
// initiales, puis essaie de retirer 0 valeurs de la collection avant
// d'afficher le résultat.
{
    cout << "\tAffichage attendu :\t1 2 3 4 5 " << endl;
    cout << "\tAffichage obtenu :\t";
    unsigned int taille = 5;
    int valeurs[] = {1, 2, 3, 4, 5};
    unsigned int n = 0;
    int retirer[] = {};
```

```
    CollectionEntiers * c = new CollectionEntiers(taille, valeurs);
    c->Retirer(n, retirer);
    c->Afficher();
    delete c;
    return true;
}
```

void Ajuster (unsigned int n)

test_11_ajuster_augmenter :

```
bool test_11_ajuster_augmenter ()
// Algorithme : Construit une nouvelle collection avec des valeurs
// initiales, puis essaie d'augmenter la taille allouée avant d'afficher
// le résultat.
{
    cout << "\tAffichage attendu :\t1 2 3 4 5 " << endl;
    cout << "\tAffichage obtenu :\t";
    unsigned int taille = 5;
    int valeurs[] = {1, 2, 3, 4, 5};
    unsigned int taille_augmentee = 10;
    CollectionEntiers * c = new CollectionEntiers(taille, valeurs);
    c->Ajuster(taille_augmentee);
    c->Afficher();
    delete c;
    return true;
}
```

test_12_ajuster_diminuer :

```
bool test_12_ajuster_diminuer ()
// Algorithme : Construit une nouvelle collection avec des valeurs
// initiales, puis essaie de diminuer la taille allouée avant d'afficher
// le résultat.
{
    cout << "\tAffichage attendu :\t1 2 " << endl;
    cout << "\tAffichage obtenu :\t";
    unsigned int taille = 5;
    int valeurs[] = {1, 2, 3, 4, 5};
    unsigned int taille_reduite = 2;
    CollectionEntiers * c = new CollectionEntiers(taille, valeurs);
    c->Ajuster(taille_reduite);
    c->Afficher();
    delete c;
    return true;
}
```

test_13_ajuster_nul :

```
bool test_13_ajuster_nul ()
// Algorithme : Construit une nouvelle collection avec des valeurs
// initiales, puis essaie de réduire la mémoire allouée à 0 avant
// d'afficher le résultat.
{
    cout << "\tAffichage attendu :\t " << endl;
    cout << "\tAffichage obtenu :\t";
    unsigned int taille = 5;
    int valeurs[] = {1, 2, 3, 4, 5};
    unsigned int taille_reduite = 0;
```

```
    CollectionEntiers * c = new CollectionEntiers(taille, valeurs);
    c->Ajuster(taille_reduite);
    c->Afficher();
    delete c;
    return true;
}
```

2. Listes détaillées

CollectionEntiers.h

```
/*************************************************************************
            CollectionEntiers  - Collection dynamique d'entiers
                             -------------------
    début                : 06/10/2015
    copyright            : (C) 2015 par B3309
*************************************************************************/

// Interface de la classe <CollectionEntiers> (fichier CollectionEntiers.h)
#if ! defined ( COLLECTION_ENTIERS_H )
#define COLLECTION_ENTIERS_H

//------------------------------------------------- Interfaces utilisées

//------------------------------------------------------------ Constantes

//------------------------------------------------------------------ Types

//------------------------------------------------------------------------
// Rôle de la classe <CollectionEntiers>
// La classe CollectionEntiers permet de gérer des collections d'entiers
// de taille quelconque fixée à la création avec réajustement automatique
// ou à la demande.
//------------------------------------------------------------------------

class CollectionEntiers
{
//----------------------------------------------------------------- PUBLIC

public:
//----------------------------------------------------- Méthodes publiques
    void Afficher () const;
    // Mode d'emploi : Affiche le contenu de la collection.
    // Les valeurs sont séparées par des espaces.
    // Contrat : <std::cout> doit exister et être accessible.

    void Ajouter (int valeur);
    // <valeur> : la valeur à ajouter à la collection
    // Mode d'emploi : Ajoute la valeur en paramètre à la collection.
    // Contrat : <valeur> doit être un entier.

    void Retirer (unsigned int n, int retirer[]);
    // <n> : le nombre de valeurs à supprimer
    // <retirer> : la liste des valeurs à retirer de la collection
    // Mode d'emploi : Retire la liste de valeurs passée en paramètre de
    // la collection, puis ajuste la collection au plus juste.
    // Contrat : <n> doit spécifier la taille de <retirer> ; <retirer>
    // doit être un tableau d'entiers à retirer de la collection.

    void Ajuster (unsigned int n);
    // <n> : la nouvelle taille de la collection
    // Mode d'emploi : Modifie la taille de la collection pour qu'elle
    // prenne la taille <n>. <n> peut être supérieur ou inférieur à la
```

```
    // taille actulle (auquel cas la collection sera tronguée à partir de
    // la <n>-ième valeur incluse).
    // Contrat : <n> doit être un entier strictement positif.

    void Reunir (CollectionEntiers * collection);
    // <collection> : la collection contenant les éléments à ajouter
    // Mode d'emploi : Ajoute les valeurs de la collection passée en
    // paramètre à celles de cette collection.
    // Contrat : <collection> doit être une CollectionEntiers bien
    // construite et initialisée.

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

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

    CollectionEntiers (unsigned int n);
    // <n> : taille initiale de la collection
    // Mode d'emploi : Construit une collection de taille <n> sans valeurs
    // de départ (allocation seule).
    // Contrat : <n> doit être un entier positif.

    CollectionEntiers (unsigned int n, int valeurs[]);
    // <n> : le nombre de valeurs initiales
    // <valeurs> : les valeurs initiales
    // Mode d'emploi : Construit une collection de taille <n> avec les
    // valeurs de départ de la collection.
    // Contrat : <n> doit spécifier la taille de <valeurs> ; <valeurs>
    // doit être un tableau d'entiers.

    virtual ~CollectionEntiers ();
    // Mode d'emploi : Supprime le tableau de valeurs que contient la
    // collection.
    // Contrat : N/A


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

    void Init (unsigned int n);
    // <n> : taille initiale à allouer
    // Mode d'emploi : Initialise les valeurs de tous les attributs.
    // Contract : <n> doit être un entier positif.

private:
//------------------------------------------------------ Méthodes privées

protected:
//----------------------------------------------------- Attributs protégés
    unsigned int alloue;
    unsigned int elements;
    int * valeurs;

private:
//-------------------------------------------------------- Attributs privés

//---------------------------------------------------------- Classes amies

//------------------------------------------------------- Classes privées
```

```
//------------------------------------------------------------ Types privés
};

//-------------------------------- Types dépendants de <CollectionEntiers>

#endif // COLLECTION_ENTIERS_H
```

CollectionEntiers.cpp

```
/*************************************************************************
            CollectionEntiers  - Collection dynamique d'entiers
                             -------------------
    début                : 06/10/2015
    copyright            : (C) 2015 par B3309
*************************************************************************/

// Réalisation de la classe <CollectionEntiers> (fichier CollectionEntiers.cpp)

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

//-------------------------------------------------------- Include système
using namespace std;
#include <iostream>

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

//------------------------------------------------------------- Constantes

//---------------------------------------------------- Variables de classe

//-------------------------------------------------------------- Types privés


//----------------------------------------------------------------- PUBLIC
//---------------------------------------------------------- Fonctions amies

//------------------------------------------------------- Méthodes publiques

void CollectionEntiers::Afficher () const
// Algorithme : Parcourt tous les éléments de <valeurs> et affiche chaque
// valeur, séparées par un espace (note : il y aura un espace à la suite
// du dernier élément, si <valeurs> est non-vide).
{
    for (unsigned int i = 0; i < elements; i++)
    {
        cout << valeurs[i] << ' ';
    }
    cout << endl;
} //----- Fin de Afficher

void CollectionEntiers::Ajouter (int valeur)
// Algorithme : Ajuste la mémoire allouée si une quantité insuffisante a
// été allouée, puis ajoute <valeur> au tableau <valeurs>.
{
    if (elements >= alloue)
    {
        Ajuster(elements + 1);
    }
    valeurs[elements] = valeur;
```

```
    elements++;

} //----- Fin de Ajouter

void CollectionEntiers::Retirer (unsigned int n, int retirer[])
// Algorithme : Si la liste est non-vide :
// 1.  Le pointeur du tableau des anciennes valeurs est copié dans un
//     pointeur temporaire <tmp> et <valeurs> est réinitialisé avec la
//     même taille.
// 2.  Une boucle itère sur chaque ancienne valeur (à partir de <tmp>) :
//     i.  Si la valeur n'est pas dans le tableau de valeurs à retirer,
//         elle est copiée dans le nouveau <valeurs>.
//     ii. Sinon, la valeur n'est pas recopiée (ce qui revient à la
//         supprimer).
// 3.  Le nouveau <valeurs> est ré-ajusté pour avoir une taille mémoire au
// plus juste.
// 4. L'ancien <valeurs> (pointé par <tmp>) est supprimé.
{
    if (n == 0)
    {
        return;
    }

    int * tmp = valeurs;
    valeurs = new int[elements];
    unsigned int k = 0;
    for (unsigned int i = 0; i < elements; i++)
    {
        bool conserver = true;
        for (unsigned int j = 0; j < n; j++)
        {
            if (retirer[j] == tmp[i])
            {
                conserver = false;
                break;
            }
        }
        if (conserver)
        {
            valeurs[k] = tmp[i];
            k++;
        }
    }
    elements = k;
    Ajuster(elements);
    delete[] tmp;
} //----- Fin de Retirer

void CollectionEntiers::Ajuster (unsigned int n)
// Algorithme : Le pointeur du tableau est copié dans un pointeur
// temporaire <tmp> et <valeurs> est ré-initialisé pour devenir un
// tableau d'entiers de taille <n>. Chaque élément de l'ancien tableau
// y est recopié, puis l'ancien tableau est supprimé. Si le nouveau
// tableau est plus petit, on garde tous les éléments, dans l'ordre dans
// lequel ils sont stockés, qui peuvent rentrer dans le nouveau tableau.
{
    alloue = n;
    elements = (elements > alloue) ? alloue: elements;
    int * tmp = valeurs;
    valeurs = new int[n];
```

```
    for (unsigned int i = 0; i < elements && i < alloue; i++)
    {
        valeurs[i] = tmp[i];
    }
    delete[] tmp;
} //----- Fin de Ajuster

void CollectionEntiers::Reunir (CollectionEntiers * collection)
// Algorithme : S'il n'y a pas assez de mémoire allouée pour contenir les
// éléments de cette collection ainsi que ceux de <collection>, la taille
// de cette collection est augmentée ; sinon elle demeure inchangée.
// Ensuite, chaque élément de <collection> est copié dans cette
// collection, et le nombre d'éléments de cette collection est ajusté.
{
    if (collection->elements + elements > alloue)
    {
        Ajuster(collection->elements + elements);
    }
    for (unsigned int i = 0; i < collection->elements; i++)
    {
        valeurs[elements + i] = collection->valeurs[i];
    }
    elements += collection->elements;
} //----- Fin de Reunir


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


//-------------------------------------------- Constructeurs - destructeur
CollectionEntiers::CollectionEntiers (unsigned int n)
// Algorithme : Initialise les attributs grâce à Init, sans opérations
// supplémentaires.
{
#ifdef MAP
    cout << "Appel au constructeur de CollectionEntiers" << endl;
#endif
    Init(n);
} //----- Fin de CollectionEntiers

CollectionEntiers::CollectionEntiers (unsigned int n, int valeurs[])
// Algorithme : Initialise les attributs grâce à Init, puis ajoute toutes
// les <valeurs> au tableau <valeurs>.
{
#ifdef MAP
    cout << "Appel au constructeur de CollectionEntiers" << endl;
#endif
    Init(n);
    for (unsigned int i = 0; i < n; i++)
    {
        Ajouter(valeurs[i]);
    }
} //----- Fin de CollectionEntiers

CollectionEntiers::~CollectionEntiers ()
// Algorithme : Libère la mémoire allouée au tableau de valeurs lors de
// l'initialisation.
{
#ifdef MAP
    cout << "Appel au destructeur de CollectionEntiers" << endl;
```

```
#endif
    delete[] valeurs;
} //----- Fin de ~CollectionEntiers


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

//----------------------------------------------------- Méthodes protégées

void CollectionEntiers::Init (unsigned int n)
// Algorithme : Initialise les attributs avec leurs valeurs initiales ;
// une collection de taille <n>, contenant 0 éléments, est créée.
{
    alloue = n;
    elements = 0;
    valeurs = new int[n];
} //----- Fin de Init

//------------------------------------------------------ Méthodes privées
```

makefile

```
CXX = g++
LN = g++
RM = @rm
ECHO = @echo

CXXFLAGS = -Wall
LNFLAGS =
COMPILEFLAG = -c
OUTPUT = -o
RMFLAGS = -f

DIR = src
INT = CollectionEntiers.h
SRC = $(INT:.h=.cpp)
OBJ = $(INT:.h=.o)
MAIN = $(DIR)/main.cpp

LIBS =
LIBPATH =
INCPATH =

BIN = tp-oo_1

ALL = all
CLEAN = clean

.PHONY: $(ALL) $(CLEAN)

$(ALL): $(BIN)

$(BIN): $(OBJ) $(MAIN)
	$(CXX) $(CXXFLAGS) $^ $(OUTPUT) $@

CollectionEntiers.o: $(DIR)/CollectionEntiers.h $(DIR)/CollectionEntiers.cpp
	$(CXX) $(CXXFLAGS) $(COMPILEFLAG) $(DIR)/CollectionEntiers.cpp

$(CLEAN):
	$(RM) $(RMFLAGS) $(OBJ) $(BIN) $(BIN).exe
```