TP C++ n°1 : Classe simple

1. Introduction

Ce premier TP d'initiation au langage C++ reste relativement simple et doit essentiellement vous permettre de développer et tester votre première classe C++ en mettant en œuvre les principes élémentaires de développement d'un logiciel et la notion d'interface et de réalisation de classe (application du Guide de Style C++ décrit dans le support de cours). Ce TP C++ insiste beaucoup sur le respect des spécifications et sur la couverture des tests (réalisation des tests unitaires de la classe à partir du plan de tests fonctionnels de la classe).
Vous allez manipuler **une seule classe** simple mais dynamique (contrainte du cahier des charges) sans faire intervenir d'héritage, ni de généricité. Ce TP met en œuvre des pointeurs, et par conséquent les principes de l'allocation dynamique de la mémoire.
Pour valider votre travail, nous allons utiliser l'outil *DOMjudge* que vous manipulez déjà dans le cadre des TP d'algorithmie ou bien un framework de test fourni (cf paragraphe 8).

2. Cahier des charges général

Votre classe doit être capable de gérer une collection d'entiers de **taille quelconque fixée à la création** avec réajustement automatique ou à la demande. Cette collection d'entiers doit correspondre à un **ensemble mathématique**. Les méthodes requises pour cette classe **`Ensemble`** sont (la liste n'est pas exhaustive mais elle est suffisante pour ce TP d'initiation) :

- Un **premier constructeur** (constructeur par défaut) pour créer une collection vide avec une dimension maximale fixée à la création ;
- Un **second constructeur** utilisant un tableau d'entiers au sens du langage C/C++. **Ce constructeur n'est pas le constructeur de copie** de la classe ;
- Une méthode **`Afficher`** pour afficher, sur la sortie standard (au sens Linux), le contenu de la collection ;
- Une méthode **`EstEgal`** pour tester l'égalité entre 2 ensembles ;
- Une méthode **`EstInclus`** pour tester l'inclusion entre 2 ensembles ;
- Une méthode **`Ajouter`** pour ajouter **un seul entier** à un ensemble (objet de votre classe). Cette opération ne doit pas ajuster la cardinalité maximale de l'ensemble si l'opération n'est pas possible ;
- Une méthode **`Ajuster`** pour modifier, à la demande, la taille maximale d'une collection ;
- Une méthode **`Retirer`** pour retirer un seul élément à un ensemble. Après l'opération, l'ensemble doit **toujours** avoir une taille mémoire **au plus juste** (occupation) ;
- Une méthode **`Retirer`** pour retirer les éléments (s'ils existent) d'un ensemble passé en paramètre dans l'ensemble qui invoque la méthode. Après l'opération, l'ensemble doit **toujours garder sa cardinalité maximale** ;
- Une méthode **`Réunir`** pour rassembler **impérativement** deux ensembles (objets de votre classe). La modification de la taille n'est tolérée que **si elle est nécessaire** ;
- Une méthode **`Intersection`** pour réaliser l'intersection entre l'objet qui invoque la méthode et celui passé en paramètre. La modification de la taille est systématique.

3. Spécification détaillée de la classe

Votre classe possède 3 caractéristiques essentielles : sa cardinalité maximale, sa cardinalité actuelle et les éléments de l'ensemble. Les cardinalités de l'ensemble sont gérées par la relation d'ordre :

`Cardinalité Maximale ≥ Cardinalité Actuelle ≥ 0`

Pour gérer les éléments de l'ensemble, **il faut obligatoirement s'appuyer sur un tableau dynamique** même si d'autres implémentations sont clairement possibles pour répondre à cette spécification détaillée.

`Ensemble ( unsigned int cardMax = CARD_MAX );`

`cardMax` est la cardinalité maximale de l'ensemble en cours construction. Ce constructeur construit un ensemble vide (c'est-à-dire ne comportant aucun élément entier) et de cardinalité maximale fixée par le paramètre **`cardMax`**. Si le paramètre est absent, c'est la constante **`CARD_MAX`** qui est utilisée avec comme valeur par défaut **`5`**. Si **`cardMax`** vaut **`0`**, l'ensemble construit est nul, c'est-à-dire que sa cardinalité actuelle et sa cardinalité maximale sont nulles.

`Ensemble ( int t [ ], unsigned int nbElements );`

`t` est un tableau d'entiers au sens du langage C/C++ et **`nbElements`** correspond au nombre d'éléments significatifs dans le tableau. Ces éléments significatifs sont obligatoirement contigus dans le tableau et en tête de ce tableau (les premiers indices). Ce nombre d'éléments peut être différent de la dimension du tableau (mais forcément inférieur ou égal).

Ce constructeur construit un ensemble d'entiers à partir des entiers contenus dans le tableau **`t`** en respectant la définition mathématique d'un ensemble. A la fin de la construction, la cardinalité maximale de l'ensemble construit est égale au paramètre **`nbElements`** et la cardinalité actuelle correspond au nombre d'éléments effectivement rajoutés dans l'ensemble.

Par définition : cardinalité actuelle ≤ cardinalité maximale.

`virtual ~Ensemble ( );`

Ce destructeur doit libérer la totalité des ressources (mémoire) occupées par l'ensemble.

`void Afficher ( void );` `// volontairement non const pour permettre le tri`

Cette méthode se charge d'afficher sur la sortie standard (l'écran) la valeur d'un objet ensemble en respectant la syntaxe suivante :

`{}`	si l'ensemble est vide ;
`{x}`	si l'ensemble est composé d'un seul élément **`x`** ;
`{x,y,z}`	si l'ensemble est composé de plusieurs éléments (**`x`**, **`y`** et **`z`** pour l'exemple).

S'il y a plusieurs éléments, ils devront toujours être affichés par ordre croissant même si, à la base, un ensemble mathématique est non ordonné. Cette exigence est liée aux tests automatiques de votre classe. Un retour à la ligne termine toujours l'affichage de l'ensemble. Pour faciliter le test de votre classe, les 2 informations suivantes sont rajoutées **avant l'affichage des valeurs** de l'ensemble, en respectant la syntaxe donnée :

`n`
`m`

où **`n`** représente la cardinalité actuelle de l'ensemble et **`m`** sa cardinalité maximale.

`bool EstEgal ( const Ensemble & unEnsemble ) const;`

`unEnsemble` donne l'ensemble qui est utilisé dans le test d'égalité avec l'ensemble qui invoque la méthode. L'ensemble **`unEnsemble`** et l'ensemble qui invoque la méthode sont égaux si et seulement s'ils ont la même cardinalité actuelle, indépendamment de leur cardinalité maximale, et si tous les éléments de l'ensemble **`unEnsemble`** sont aussi présents dans l'ensemble qui invoque la méthode. Dans ce cas, la méthode renvoie *vrai*. Si les cardinalités actuelles sont égales et qu'il existe au moins un élément de l'ensemble qui invoque la méthode qui n'appartient pas à l'ensemble **`unEnsemble`** alors l'égalité n'est pas satisfaite et la méthode renvoie *faux*. Ce cas est indépendant des valeurs des cardinalités maximales. Si les cardinalités actuelles sont différentes, l'égalité des ensembles n'est pas vérifiée et la méthode renvoie *faux*.

`crduEstInclus EstInclus ( const Ensemble & unEnsemble ) const;`

`unEnsemble` est l'ensemble utilisé pour vérifier l'inclusion. Cette méthode vérifie si l'ensemble qui invoque la méthode est inclus dans l'ensemble **`unEnsemble`** (non inclusion, inclusion large ou inclusion stricte). Si les 2 ensembles sont égaux, l'inclusion est vérifiée. Dans ce cas, la méthode renvoie **`INCLUSION_LARGE`**. S'il existe au moins un élément de l'ensemble qui invoque la méthode qui ne se retrouve pas dans **`unEnsemble`**, alors l'inclusion n'est pas vérifiée et la méthode renvoie **`NON_INCLUSION`**. Si tous les éléments de l'ensemble qui invoque la méthode se retrouvent dans **`unEnsemble`** et que les 2 ensembles ne sont pas égaux (premier cas de figure), alors l'inclusion est strictement vérifiée et la méthode renvoie **`INCLUSION_STRICTE`**.

Pour réaliser cette méthode, il faut définir une énumération composée, **dans cet ordre**, de **`NON_INCLUSION`**, **`INCLUSION_LARGE`** et **`INCLUSION_STRICTE`**.

`crduAjouter Ajouter ( int aAjouter );`

`aAjouter` est l'élément entier à rajouter à l'ensemble, si cela est possible. L'élément **`aAjouter`** est rajouté à l'ensemble, si cela est nécessaire et possible. En effet, la cardinalité maximale de l'ensemble doit rester inchangée lors de cette opération. La méthode renvoie **`DEJA_PRESENT`**, si l'élément **`aAjouter`** appartient déjà à l'ensemble (l'ajout devient inutile). Dans tous les cas de figure, **`DEJA_PRESENT`**

l'emporte sur **PLEIN**, si les deux conditions sont vraies simultanément. La méthode renvoie **PLEIN**, si l'élément **aAjouter** n'existe pas déjà dans l'ensemble et qu'il n'y a plus de place dans l'ensemble. La méthode renvoie **AJOUTE**, si l'élément **aAjouter** n'existe pas déjà dans l'ensemble et qu'il y a encore de la place dans l'ensemble. Dans ce dernier cas, la cardinalité actuelle est mise à jour pour refléter l'ajout de l'élément à l'ensemble.

Pour réaliser cette méthode, il faut définir une énumération composée, **dans cet ordre**, de **DEJA_PRESENT**, **PLEIN** et **AJOUTE**.

```
unsigned int Ajuster ( int delta );
```

delta donne le nombre d'éléments du réajustement. Si **delta** est strictement positif, l'ensemble est agrandi du nombre d'éléments défini par **delta** : c'est un agrandissement de l'ensemble. Si **delta** est strictement négatif, l'ensemble est réduit du nombre d'éléments défini par **delta** (dans les limites possibles) : c'est une réduction de l'ensemble. Cette réduction ne peut pas s'accompagner de perte d'éléments dans l'ensemble. Si **delta** est nul, l'opération est sans effet et la valeur de retour est la cardinalité maximale initiale de l'ensemble. Dans tous les cas de figure, la valeur de retour est la nouvelle cardinalité maximale de l'ensemble.

```
bool Retirer ( int element );
```

element est l'élément entier à retirer de l'ensemble, si cela est possible (existence). L'élément entier est retiré de l'ensemble s'il est présent. La méthode renvoie *vrai*, si l'élément a été retiré de l'ensemble. La méthode renvoie *faux*, si le retrait a échoué. Dans tous les cas de figure, la cardinalité maximale sera égale à la cardinalité actuelle (réajustement de l'ensemble), même si aucun élément n'est retiré de l'ensemble.

```
unsigned int Retirer ( const Ensemble & unEnsemble );
```

unEnsemble contient les éléments qu'il faut retirer à l'ensemble qui invoque la méthode, si cela est possible. Cette méthode retire les différents éléments de l'ensemble **unEnsemble** de l'ensemble qui invoque la méthode (existence des éléments). La méthode renvoie 0, si aucun élément n'a été retiré de l'ensemble. La méthode renvoie une valeur > 0, si au moins un élément de l'ensemble **unEnsemble** existe bien dans l'ensemble qui invoque la méthode et qu'il a été bien retiré de l'ensemble. En fait, la valeur rendue correspond au nombre d'éléments effectivement retirés de l'ensemble ⇒ diminution de la cardinalité actuelle de cette valeur. Dans tous les cas de figure (retrait ou pas), la cardinalité maximale restera inchangée (pas de réajustement au plus juste).

```
int Reunir ( const Ensemble & unEnsemble );
```

unEnsemble est l'ensemble qui va être utilisé pour effectuer la réunion avec l'ensemble qui invoque la méthode. Cette méthode rajoute à l'ensemble qui invoque la méthode tous les éléments de l'ensemble **unEnsemble** qui ne sont pas déjà présents dans l'ensemble courant. La méthode renvoie une valeur strictement négative si l'ensemble qui invoque la méthode a été réajusté pour contenir la réunion. Dans ce cas, le réajustement se fait au plus juste. La valeur absolue rendue correspond au nombre d'éléments de l'ensemble **unEnsemble** effectivement rajoutés. La méthode renvoie une valeur strictement positive si l'ensemble qui invoque la méthode n'a pas été réajusté pour contenir la réunion. Dans ce cas, la valeur rendue correspond au nombre d'éléments de l'ensemble **unEnsemble** effectivement rajoutés. La méthode renvoie 0 si aucun élément de l'ensemble **unEnsemble** n'a été rajouté à l'ensemble courant pour réaliser l'union. Autrement dit, l'ensemble **unEnsemble** est inclus dans l'ensemble courant. Dans ce dernier cas, il n'y a pas de réajustement de la cardinalité maximale.

```
unsigned int Intersection ( const Ensemble & unEnsemble );
```

unEnsemble est l'ensemble qui va être utilisé pour effectuer l'intersection avec l'ensemble qui invoque la méthode. Cette méthode modifie l'ensemble qui invoque la méthode en retenant uniquement les éléments en commun entre les 2 ensembles (celui qui invoque la méthode et celui qui est en paramètre). La méthode renvoie le nombre d'éléments supprimés dans l'ensemble qui invoque la méthode pour bâtir l'intersection. Après l'opération d'intersection et quel que soit le cas de figure, l'ensemble est réajusté au plus juste.

4. Réalisation

Comme votre classe doit être capable de gérer des ensembles de **taille quelconque** (mais fixée à la création et avec des ajustements éventuels ultérieurs), il est évident que sa réalisation exige l'utilisation de l'allocation dynamique de la mémoire. **Cette gestion dynamique nécessite la plus grande concentration et rigueur : il y a le pointeur et il y a l'objet pointé !** Durant le codage, **on doit respecter le *Guide de Style C++*** (squelette des interfaces et réalisations, convention d'écriture...) (cf. *Support de cours C++*).
L'accent est mis sur la qualité de votre programmation (sa simplicité) et sur le respect de la spécification détaillée de la classe. Il faut construire une classe **`Ensemble`** répondant parfaitement aux besoins du client !

Attention : contrainte de réalisation

L'utilisation d'une quelconque fonction issue d'une quelconque bibliothèque standard du C / C++, hormis la bibliothèque de manipulation des flots d'entrées / sorties **`iostream`**, est **rigoureusement interdite**.
`iostream` ne doit servir qu'à faire des **`cin`**, **`cout`** ou **`cerr`** !
Le compilateur g++ du serveur *DOMjudge* ne gère pas le mot clé C++ **`nullptr`**. Il faut encore utiliser la constante **`NULL`**.

5. Du plan de tests fonctionnels à sa réalisation

En principe, la spécification détaillée est exhaustive et il ne devrait plus y avoir de comportements à interpréter ou à deviner.
À partir de cette spécification détaillée de la classe **`Ensemble`**, il reste à construire le plan de tests fonctionnels nécessaire à la validation de votre réalisation. Pour chacune des méthodes, il faut clairement dire ce qu'il faut tester en fonction de sa description détaillée et des choix effectués et, en aucun cas, comment il faut le tester (réalisation des jeux d'essai).
Par exemple, « *Vérifier que la construction d'un ensemble utilisant le constructeur par défaut sans paramètre fournit un ensemble vide (c'est-à-dire de cardinalité actuelle nulle) et de cardinalité maximale, la valeur retenue par défaut (c'est-à-dire CARD_MAX)* » définit un test fonctionnel. À ce stade, on ne précise pas comment le test va être effectivement réalisé : nombre d'objets à créer, méthodes à invoquer et dans quel ordre, résultats attendus…
Dans un deuxième temps, il sera possible de passer du plan de tests fonctionnels à sa réalisation en appliquant les principes brièvement résumés ci-dessous pour le constructeur par défaut :

```
static void testConstructeurDefaut1 ( )
{
      // Réalisation du premier test fonctionnel – Jeu d'essai
      // Avec une mise en évidence du résultat attendu (sous forme de commentaire)
      // Isolation de chaque test pour minimiser les effets de bords
}

// Autres tests du constructeur par défaut

static void testConstructeurDefaut ( )
{
      testConstructeurDefaut1 ( );
      // testConstructeurDefaut2 ( );
}

int main ( )
{
      testConstructeurDefaut ( );
      return 0;
}
```

Dans tous les tests réalisés, la méthode **`Afficher`** sera utilisée (comparaison des sorties attendues et des sorties obtenues). Cette méthode ne fera pas l'objet d'un test unitaire séparé.

Note :

Le destructeur de la classe et la bonne gestion de la mémoire dynamique seront validés par l'outil **`valgrind`**. (cf. TP IF-3-OP-1). Au fur et à mesure de l'avancée de votre réalisation, il faudra s'assurer de l'absence de fuite de mémoire et de la bonne gestion de la mémoire. Cette vérification n'est pas prise en compte par un test unitaire de la classe.

6. Validation sous *DOMjudge*

Le serveur *DOMjudge* utilisé dans le cadre de ce *TP C++ Classe Simple* pour valider votre classe se trouve à l'adresse : **http://servif-cpp.insa-lyon.fr/domjudge/public**. Avant de valider votre classe avec cet outil, il est impératif de construire son propre module de tests unitaires et de réaliser chacun des tests unitaires localement (gain de temps).

Les 10 méthodes de la classe **`Ensemble`** concernées par un test unitaire sont :

```
Ensemble ( unsigned int cardMax = CARD_MAX );                    // TU01
Ensemble ( int t [ ], unsigned int nbElements );                 // TU02
bool EstEgal ( const Ensemble & unEnsemble ) const;              // TU03
crduEstInclus EstInclus ( const Ensemble & unEnsemble ) const;// TU04
crduAjouter Ajouter ( int aAjouter );                            // TU05
unsigned int Ajuster ( int delta );                              // TU06
bool Retirer ( int element );                                    // TU07
unsigned int Retirer ( const Ensemble & unEnsemble );            // TU08
int Reunir ( const Ensemble & unEnsemble );                      // TU09
unsigned int Intersection ( const Ensemble & unEnsemble );       // TU10
```

Le destructeur de la classe **`Ensemble`** ne fera pas l'objet de tests spécifiques. Nous nous contenterons des diagnostics de l'outil **`valgrind`** pour évaluer la bonne gestion dynamique de la mémoire.
De même, la méthode **`Afficher`** ne sera pas explicitement testée. Nous effectuerons des tests joints avec les autres méthodes de la classe pour vérifier leur comportement et nous profiterons de ces tests pour valider la méthode **`Afficher`**.

Pour chaque test unitaire **`TUi`** à effectuer sous *DOMjudge* (problème dans le vocabulaire *DOMjudge*), vous disposez d'un langage propre **`C++ - TUi`** (**`i`** correspond au numéro du test unitaire (de **`01`** à **`ALL`**)), qui correspond à notre plan de tests fonctionnels pour chacune des méthodes à tester de la classe **`Ensemble`**. Si votre réalisation de la classe **`Ensemble`** est conforme aux spécifications, alors le test unitaire devrait bien se passer. Dans le cas contraire, il y aura probablement un échec dans l'outil *DOMjudge*. Pour éviter toute divergence entre la sortie de référence et la sortie de votre programme, il est impératif de respecter les 2 règles suivantes :

1. Chaque **`cout`** de votre réalisation doit impérativement utiliser la séquence **`"\r\n"`** à la place de **`endl`**. Cette contrainte est déjà connue puisqu'elle existait déjà dans les TD / TP d'algorithmie ;
2. Lorsque vous soumettez votre classe à l'outil *DOMjudge*, il ne doit pas subsister de **`cout`** parasites : autrement dit, les seuls **`cout`** autorisés se trouvent dans la méthode **`Afficher`** de la classe **`Ensemble`**.

Pour soumettre un test unitaire à *DOMjudge* (problème dans le vocabulaire *DOMjudge*), il faut fournir les sources de votre classe (**`Ensemble.h`** et **`Ensemble.cpp`**) et choisir le bon langage **`C++ - TUi`** correspondant au test unitaire qui vous intéresse. Par conséquent, les tests unitaires se passent complètement en aveugle. A priori vous ne connaissez ni les **`IN`**, ni les **`OUT`** d'un test unitaire.

7. Récupération des informations

Pour réaliser ce TP, il faut utiliser les squelettes de classe présentés et utilisés lors des cours C++. Ils sont directement accessibles sous *Moodle*.

8. Autre outil de Validation : *le framework de tests*

Ce *framework* sert également à valider votre classe. Il est plus *gentil* que *DOMjudge* puisqu'il vous donne, pour chaque test **`TUi`**, le résultat attendu que vous pourrez comparer avec votre résultat obtenu. En cas d'échec, cela peut aider à comprendre et découvrir le problème. Les tests n'opèrent plus complètement en aveugle.
Pour utiliser ce *framework* :

a. Il faut copier l'archive Linux **`FRAMEWORK-TEST.tar.gz`**, accessible à partir de moodle, dans votre répertoire de travail ;
b. Il faut décompresser l'archive en utilisant la commande Linux **`tar`** avec la syntaxe suivante :
`tar xzvf FRAMEWORK-TEST.tar.gz`
c. Dans le répertoire **`FRAMEWORK-TEST`** obtenu, il faut copier votre interface et votre réalisation de classe (**`Ensemble.h`** et **`Ensemble.cpp`**) ;
d. Pour réaliser le test unitaire **`TUi`**, il faut se placer dans le répertoire **`Tests`** et lancer la commande **`./test.sh Testi`** ;

Avec cet outil, vous ne connaissez toujours pas les **`IN`** (jeux d'essai en entrée) mais vous pourrez voir les **`OUT`** obtenus et les **`OUT`** attendus, pour chaque test unitaire.
Pour valider l'ensemble des tests unitaires, il est possible de lancer le script shell **`./mktest.sh`**.
Avant de valider votre classe avec ce *framework,* il est également vivement recommandé de construire son propre module de tests unitaires et de réaliser chacun des tests unitaires localement (gain de temps).