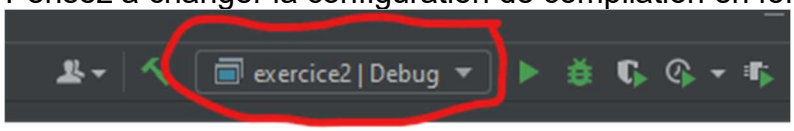
Module R3.04 – TP02
2 séances de 1h30

Préambule

- Téléchargez l'archive TP02.zip et décompressez son contenu dans votre répertoire R3-04
- Ouvrez dans CLion le projet TP02
- Pensez à changer la configuration de compilation en fonction de l'exercice :



Exercice 1. Classe EntierContraint (suite) : Surcharge d'Opérateurs

On continue l'exercice 2 du TP précédent. L'objectif est de faire en sorte qu'un objet de la classe **`EntierContraint`** (fournie dans ce TP) puisse être manipulé comme si c'était un **`int`**. Pour cela nous allons compléter la classe en lui rajoutant trois **opérateurs**. C'est possible car en C++ il est possible de surcharger tous les opérateurs du langage (liste des opérateurs : http://www.cplusplus.com/doc/tutorial/operators/) pour programmer leur comportement lorsque leurs opérandes sont des types/classes que l'on a définis nous-même.

Vous allez surcharger 3 opérateurs : l'opérateur de **cast** qui permettra au compilateur de convertir un **`EntierContraint`** en **`int`** quand c'est nécessaire, et les opérateurs **`<<`** et **`>>`** afin de pouvoir écrire/lire des objets de classe **`EntierContraint`** comme on le ferait avec des **`int`**.

Question 1.1. Surcharger l'opérateur de `cast` (conversion de type)

Cet opérateur est une **méthode** qu'il faut ajouter à la classe **`EntierContraint`**. Son prototype est :

```
EntierContraint::operator int() const ;
```

- C'est une méthode qui doit renvoyer un résultat de type **`int`**.
- On ne spécifie pas le type de retour de cette méthode car le nom de l'opérateur définit déjà le type de résultat qui sera renvoyé.
- Cette méthode renverra simplement en résultat la **valeur** de l'entier contraint.
- Vous pouvez l'implémenter sous former de méthode **inline** (Cours 1 - Chapitre 2) dans le fichier **EntierContraint.h**

Question 1.2. Surcharger l'opérateur << pour les flux de sortie

Attention, cet opérateur est une **fonction** (ce n'est pas une méthode d'instance). Son prototype est :

```
std::ostream & operator << (std::ostream & sortie, const EntierContraint & ec)
```

- Cette fonction reçoit en paramètre le flux (**`sortie`**) sur lequel on souhaite écrire un entier contraint (**`ec`**) passé par référence constante
- Elle devra écrire la valeur de l'entier contraint sur le flux
- Elle doit renvoyer en résultat le flux sur lequel on vient d'écrire, afin de pour pouvoir chaîner l'opérateur en écrivant par exemple : **`cout << ec1 << ec2`**
- Vous déclarerez le prototype dans le fichier **`EntierContraint.h`**, **après** la déclaration de la classe **`EntierContraint`**, et vous implémenterez la fonction dans le fichier **`cpp`**

Question 1.3. Surcharger l'opérateur >> pour les flux d'entrée

Attention, cet opérateur est aussi une **fonction** (ce n'est pas une méthode !). Son prototype est :

```
std::istream & operator >> (std::istream & entree, EntierContraint & ec)
```

- Cette fonction reçoit en paramètre le flux (**entree**) sur lequel on souhaite lire la un entier contraint (**ec**) passé par référence (non constante !)
- Elle devra lire un entier sur le flux et affecter cet entier à la valeur de l'entier contraint
- Elle doit renvoyer en résultat le flux sur lequel on vient de lire (pour pouvoir chaîner l'opérateur en écrivant par exemple : **cin >> ec1 >> ec2**
- Vous déclarerez le prototype dans le fichier **EntierContraint.h**, après la déclaration de la classe **EntierContraint**, et vous implémenterez la fonction dans le fichier **cpp**

Question 1.4. Tester les opérateurs

Dans le fichier exercice1.cpp, écrivez quelques tests pour vérifier que les opérateurs fonctionnent.

Vérifiez que vous pouvez :

- Affecter un **EntierContraint** à un **int** en faisant une conversion explicite :
  ```
  EntierContraint ec(5,0,100) ; int i ; i=int(ec) ; // ou i=(int)ec ;
  ```
- L'opérateur de **cast** sera même appelé implicitement par le compilateur :
  ```
  i=ec ;   // sera traduit par le compilateur en : i = int(ec) ;
  i=1+ec ; // sera traduit par le compilateur en : i = 1+int(ec) ;
  ```
- Lire la valeur d'un entier contraint : `cin >> ec ;`
- Ecrire la valeur d'un entier contraint : `cout << ec ;`

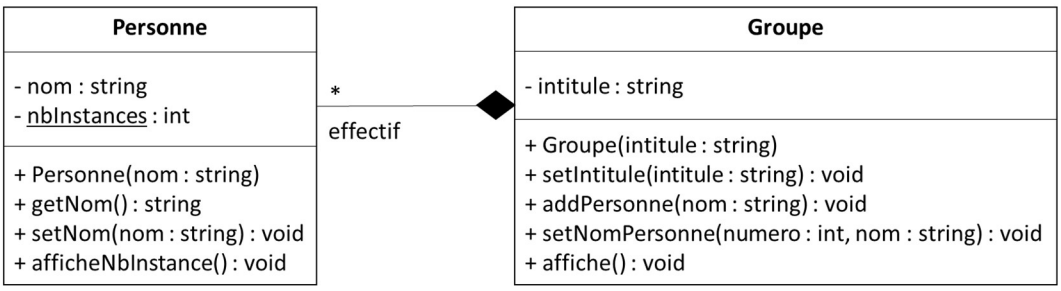
On aimerait aussi pouvoir affecter un **int** à un **EntierContraint**, par exemple : **ec=i ;**

- Est-ce que, selon votre intuition, cela va fonctionner ?
- Si non, que faudrait-il faire pour que ça marche ?
- Si oui, quel est le mécanisme mis en œuvre ? (Utilisez le debugger si besoin)

Exercice 2 - Allocation, Forme Canonique de Coplien

- *En préambule à cette question, (re)lisez le Cours 1 – Chapitre 3 & 4*

On a demandé à un débutant en C++, qui a fait du Java auparavant, de modéliser des **Groupes** composés de **Personnes**, selon le schéma UML suivant :



Remarquez que le schéma précise (on reverra cela plus tard) qu'un Groupe est **composé** de Personnes, ce qui signifie que les Personnes appartiennent à leur Groupe et à lui seul : elles ne peuvent pas appartenir à deux groupes différents en même temps.

Remarquez également que la classe **`Personne`** dispose d'une méthode de classe permettant d'afficher le nombre de Personnes instanciées.

Avec les classes fournies (allez lire leur code), on a écrit un programme de test (exercice2.cpp) qui :

- Instancie un groupe « A » composé de 3 personnes : {Emmanuel, Brigitte, Edouard}
- Instancie un groupe « B », par **recopie** du groupe A et change le nom du 2ème membre en « Line ». On devrait donc avoir 6 personnes instanciées, un groupe A inchangé et un groupe B composé de {Emmanuel, Line, Edouard}
- Instancie un groupe « C », auquel on **affecte** le groupe B et on change le nom du 3ème membre en « François ». On devrait donc avoir maintenant 9 personnes instanciées, des groupes A et B inchangés et un groupe C contenant {Emmanuel, Line, François}.
- Supprime tous les groupes créés, ce qui devrait avoir pour effet de supprimer toutes les Personnes (composition !).

Hélas le résultat n'est pas celui espéré. Lancez le programme et voyez par vous-même…

Question 2.1. Expliquez ce qui se passe

- Faites un schéma des variables et des objets instanciés (**`Groupes`** et **`Personnes`**) au fur et à mesure de l'exécution, en précisant bien ce qui est dans la **pile** et ce qui est dans le **tas**
- Si besoin, utilisez le debugger pour voir les adresses des objets en mémoire
- Vous devriez comprendre que le problème vient de la classe **`Groupe`**…

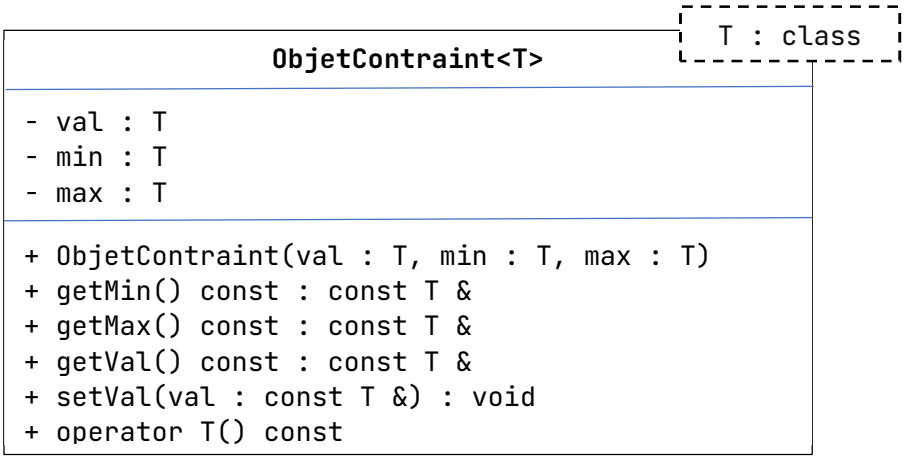
Question 2.2. Corrigez le problème

- Pour que notre code fonctionne correctement, il n'y a rien à modifier…
- Vous devez juste **compléter** la classe **`Groupe`** pour qu'elle soit sous **forme canonique de Coplien**
- Lorsque ce sera fait, relancer le programme et vérifiez qu'il fonctionne correctement

Exercice 3 - Template `ObjetContraint<T>`

- *En préambule à cette question, (re)lisez le Cours 2 - chapitre 5 (templates)*

On souhaite rendre générique la classe **`EntierContraint`** que l'on a développée au TP01. On veut donc développer un *template* de classe **`ObjetContraint<T>`** qui nous permettra de manipuler une valeur de type **`T`** en garantissant qu'elle est toujours comprise entre une valeur minimum (de type **`T`**) et une valeur maximum (de type **`T`**) et en levant une exception lorsque ce n'est pas le cas.
La spécification UML de cette classe générique est la suivante :



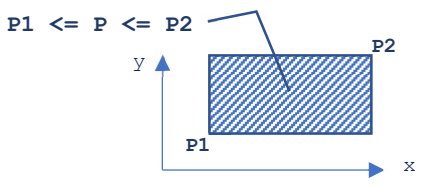
Question 3.1. Implémenter et tester le template `ObjetContraint<T>`

- Codez le template **`ObjetContraint<T>`** dans le fichier **`ObjectContraint.h`**
- Les méthodes seront implémentées dans le fichier **`.h`** (c'est nécessaire pour un *template*). Vous devez faire en sorte que seul l'opérateur « inférieur à » (**`<`**) soit utilisé pour comparer **`val`** à **`min`** et **`max`**. Ainsi notre *template* pourra être instancé avec n'importe quel type **`T`** pourvu que ce type dispose l'opérateur **`<`**.
- Vous pouvez récupérer et adapter le code de votre classe **`EntierContraint`** : le code de la classe **`ObjetContraint`** sera similaire ! Il suffit (presque) de remplacer partout **`int`** par **`T`** !
- Dans le fichier **`exercice3.cpp`**, recopiez les tests que vous aviez écrits pour la classe **`EntierContraint`** et remplacez partout **`EntierContraint`** par **`ObjetContraint<int>`**

  Vos tests devraient fonctionner sans autre modification et donner bien sûr le même résultat

Question 3.2. Instancier le template `ObjetContraint<T>` avec la classe `Point` (fournie)

- Complétez votre fichier **`exercice3.cpp`** pour :
  - Demander à l'utilisateur de saisir un point **`P1`**
  - Demander à l'utilisateur de saisir un point **`P2 (>= P1)`**
  - Demander à l'utilisateur les coordonnées d'un point **`P`** et afficher un message disant si le point **`P`** se trouve dans le rectangle délimité par **`P1`** et **`P2`**



- Vous l'aurez compris, il va falloir utiliser un **`ObjetContraint<Point>`** et se servir de l'exception qu'il est susceptible de lever.

  Mais pour pouvoir instancier le *template* **`ObjetContraint`** avec la classe **`Point`**, il faudra au préalable doter celle-ci de l'opérateur **`<`** en lui ajoutant la **méthode** :

  `bool Point::operator < (const Point &p) const`

Exercice 4 - Template Conteneur<T>

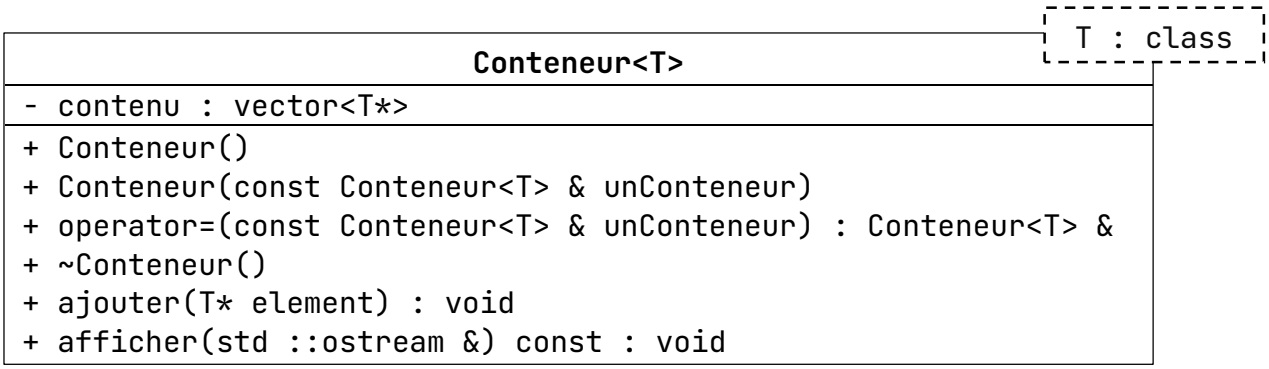
- *En préambule à cette question, (re)lisez le Cours 2 - Chapitre 6 (STL)*

Toujours dans l'optique de se doter d'outils pour un prochain TP, on souhaite développer une classe générique **`Conteneur<T>`** qui doit permettre de stocker des objets de type quelconque et de permettre à un utilisateur de choisir un élément parmi le contenu de conteneur.

Le *template* **`Conteneur<T>`** doit permettre :

- D'ajouter à un conteneur un objet de type **`T`** **qui devra avoir été alloué dynamiquement (`new`)**
- D'afficher sur un flux de sortie une liste, numérotée à partir de 1, de tous les éléments de type **`T`** contenus dans le conteneur. On supposera que la classe **`T`** dispose de l'opérateur **`<<`** pour pouvoir afficher sur le flux un objet de type **`T`**
- De demander à l'utilisateur de choisir (sur un flux d'entrée) le numéro d'un élément de type **`T`** qui existe dans le conteneur (on vérifiera que le numéro est bien compris entre **`1`** et le nombre d'élément du conteneur). On reverra en résultat l'élément choisi (par référence) ou on lèvera une exception si l'élément choisi n'existe pas.
- On utilisera dans le conteneur un attribut de type **`vector<T*>`** pour stocker les pointeurs des éléments ajoutés au conteneur.
- Les éléments ajoutés au conteneur sont réputés lui appartenir et il devra donc se charger de leur destruction à la fin de son existence, et de leur clonage en cas de recopie ou d'affectation. Bref, le conteneur doit être sous **forme canonique de Coplien** !

Spécification :



Travail à réaliser :

- Développez le *template* **`Conteneur<T>`** dans le fichier **`Conteneur.h`**
- Testez le conteneur dans le fichier **`exercice4.cpp`**. Vous pouvez par exemple ajouter des objets de type **`string`** dans le conteneur et demander à l'utilisateur d'en choisir une.