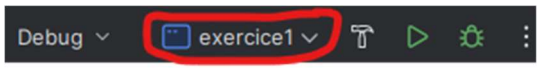
Module R3.04 – TP01
2 séance de 1h30

Préambule

- Créez un répertoire **`R3-04`** dans votre répertoire personnel
- Téléchargez l'archive **`TP01.zip`** fournie sur Chamilo et décompressez le dossier **`TP01`** dans le répertoire **`R3-04`**
- Pour lancer **Clion**, tapez dans un shell la commande : **`clion &`**
- Lors du premier lancement de Clion, vous devrez vous authentifier sur le site de Clion pour bénéficier la licence éducation gratuite que vous avez demandée (cf le mail qui vous a été envoyé en début de semaine). Si vous n'avez pas encore votre licence éducation, vous pouvez vous identifier avec une adresse mail « personnelle » (pas celle de l'UGA !) pour bénéficier d'une période d'essai de 30 jours… en attendant votre licence éducation
- Depuis Clion, ouvrez le projet **`TP01`** que vous avez téléchargé
- Dans le projet **`TP01`**, il y a un dossier par exercice (cf onglet Project), avec des fichiers à compléter. Lorsque vous changez d'exercice, changez également la « configuration » de compilation en fonction de l'exercice choisi, dans la barre de menu à droite :



Exercice 1. Classe Point

Question 1.1. Ecrire la classe Point - Cours 1, chapitres 1 et 2

Point
- nom : string - x : int - y : int
+ Point() + Point(nom : string, x : int, y : int) + ~Point() + getNom() : string + setNom(nom : string) : void … // autres getters/setters + saisir(istream & entree) : void + afficher(ostream & sortie) : void

Ecrire la spécification (fichier **Point.h**) puis l'implémentation (fichier **Point.cpp**) de la classe ci-dessus qui permet de représenter un point du plan caractérisé par son nom et ses coordonnées x et y.

Consignes :

- Vous veillerez à faire un bon usage du mot-clé **`const`** pour qualifier les méthodes d'instance qui ne modifient pas l'objet.
- Réfléchissez à la façon de bien déclarer les paramètres de type string qui sont des objets et doivent donc être passés par **référence** pour plus d'efficacité
- Le **constructeur par défaut** devra initialiser le nom et les coordonnées d'un point avec les constantes de classe (**`NOM_DEF`**, **`X_DEF`**, **`Y_DEF`**) que l'on vous a fournies. Le(s) constructeur(s) de cette classe devront afficher un message indiquant qu'un point a été instancié et son nom
- Le destructeur de cette classe devra afficher un message indiquant qu'un point a été supprimé et le nom de ce point
- La méthode **`saisir`** devra permettre de saisir, sur un flux d'entrée (**`std::cin`** par défaut), le nom du point et ses coordonnées x et y
- La méthode **`afficher`** devra permettre d'afficher sur un flux de sortie (**`std::cout`** par défaut), le nom du point et ses coordonnées
- Pensez à inclure les bibliothèques nécessaires (**`string`** et **`iostream`**) et rappelez-vous que les classes et objets qu'elles vous proposent sont définis dans l'espace de nom **`std`**
- **Corriger jusqu'à ce qu'il n'y ait plus aucune erreur à la compilation.**
  Pour compiler une classe seule, utilisez Ctrl+Maj+F9 (ou menu Build/Recompile…)

Question 1.2. Tester la classe Point : Allocation automatique - Cours : chapitre 3

Il faut maintenant instancier des objets de la classe **`Point`** pour pouvoir tester notre classe. Pour tester une classe, il faut évidemment exécuter toutes ses méthodes et vérifier qu'elles fonctionnent correctement. Un bon test est un test qui couvre 100% du code. Cela signifie que lorsque le test sera exécuté, **toutes** les lignes de code de la classe testée auront été exécutées au moins une fois.
Dans le fichier **`exercice1.cpp`** (qui contient la procédure main de cet exercice), écrivez une procédure **`testClassePoint`** qui reçoit en paramètre une chaîne **`nom`**, un entier **`x`** et un entier **`y`**. La procédure doit créer par **allocation automatique** un objet **`p`** de classe **`Point`** avec les paramètres **`(nom,x,y)`** et doit vérifier que la classe **`Point`** fonctionne correctement.

Consignes :

- Lisez le chapitre 3 du cours, « Cycle de Vie des Objets » transparents 35 à 49
- Votre procédure de test devra tester **chaque** méthode de la classe en affichant un message expliquant ce qui est testé et le résultat obtenu. Par exemple, pour tester que le getter **`getX()`** fonctionne, en sachant que l'abscisse du point **`p`** que l'on a créé doit valoir **`x`**, on écrirait le code suivant :

```
cout << "Test de getX()"                    << endl
     << "Valeur attendue : " << x           << endl
     << "Valeur obtenue  : " << p.getX()    << endl
     << (p.getX()==x ? "Succès" : "Echec")  << endl;
```

- Dans la procédure **`main`**, appelez la fonction **`testClassePoint`** plusieurs fois avec différentes valeurs.
- Corrigez votre fichier **`exercice1.cpp`** jusqu'à ce qu'il compile correctement (touche **F9**)
- Lorsque c'est le cas, lancez votre programme (touche **F6**)
- Lisez attentivement la trace produite pour comprendre ce qu'il se passe

Question 1.3. Tester la classe Point : Allocation dynamique - Cours : chapitre 3

Ré-écrire la procédure de test en réalisant l'instanciation de l'objet testé par allocation dynamique.

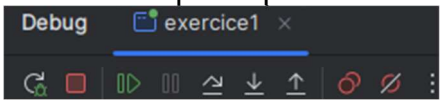
Consignes :

- Dans le fichier **`exercice1.cpp`**, dupliquez la procédure **`testClassePoint`** en **`testClassePointDynamique`** et modifiez cette dernière pour que la création du **`Point p`** soit faite par allocation dynamique
- Modifiez votre procédure **`main`** pour utiliser **`testClassePointDynamique`** et vérifiez que tout fonctionne de la même manière. Les traces doivent être identiques, à la ligne près ! Si ce n'est pas le cas, réfléchissez à ce qui se passe…

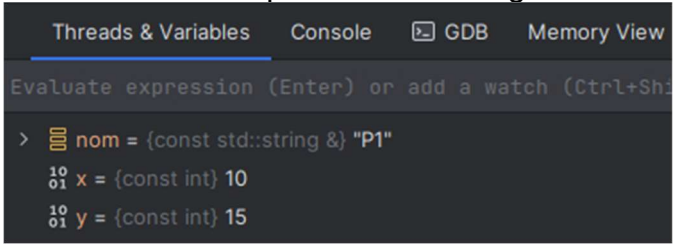
Question 1.4. Se familiariser avec le débugger (intuitif… sinon demandez à votre enseignant)

Utilisez le debugger de Clion pour (re)découvrir comment on exécute un programme pas à pas et comment on inspecte les variables d'un programme. Pour lancer le debugger : **Alt-Maj-F9**

L'exécution pas à pas se contrôle avec le panneau qui apparaît dans la fenêtre « Debug » :



Les variables s'inspectent dans l'onglet « Variables » qui apparaît dans le panneau du bas :



Exercice 2 - Classe EntierContraint

Dans un prochain TP, nous aurons besoin de manipuler la notion d'entier contraint. Nous allons donc développer en C++ une classe pour représenter cette notion.
Un entier contraint est un entier dont la valeur doit toujours être comprise dans un intervalle [min..max] qui est précisé lors de la construction de l'objet et qui ne sera plus modifié ensuite, la valeur de l'entier contraint pouvant, elle, changer.

Opérations à réaliser sur un entier contraint :

- Le construire en spécifiant sa valeur initiale ainsi que l'intervalle [min..max], ou alors, sans rien préciser (constructeur par défaut), le construire avec une valeur égale à 0 et contraint sur le plus grand intervalle possible pour un entier
- Consulter sa valeur, son min et son max
- Modifier sa valeur
- Saisir uniquement sa valeur sur un flux d'entrée
- Afficher uniquement sa valeur sur un flux de sortie

Le diagramme UML de cette classe serait le suivant :

EntierContraint
- min : int - max : int - val : int
+ EntierContraint(valeur : int , min : int, max : int) + getMin() : int + getMax() : int + getVal() : int + setVal(val : int) : void + saisir(entree : istream &) : void + afficher(sortie : ostream &) : void

Consignes :

- Ecrire la **déclaration** de la classe dans le fichier **`EntierContraint.h`**
  - Réfléchissez bien aux méthodes qui doivent être déclarées constantes
- Ecrire la **définition** de cette classe dans le fichier **`EntierContraint.cpp`**
  - Toutes les méthodes de la classe doivent être implémentées de façon à garantir en permanence la cohérence des données, à savoir : min ≤ valeur ≤ max. Lorsque cette cohérence ne peut pas être garantie, une exception de type **`char const*`** doit être levée (**`throw "erreur …").`**
- Ecrire un **programme principal** (**`exercice2.cpp`**) pour tester votre classe.
  - Votre programme de test doit permettre de couvrir 100% du code de votre classe
  - Chaque test de votre programme doit afficher : la méthode testée, le résultat attendu, le résultat obtenu et si le test a réussi ou échoué
  - Lorsque l'on veut vérifier qu'une exception a bien été levée comme prévu, il faut réaliser un test du type :

```
try {
  instruction_qui_doit_lever_une_exception ;
  cout << "Echec : pas d'exception levée)" << endl ;
}
catch (char const * erreur) {
  cout << "Succès : exception levée : " << erreur << endl ;
}
```