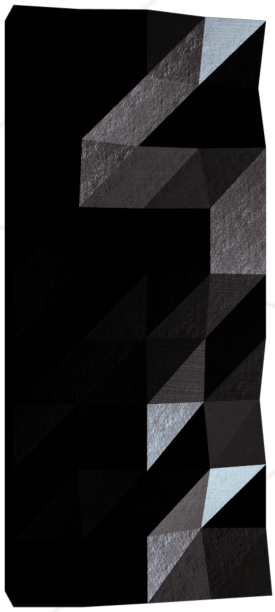

C++ - Module 05

Répétitions et exceptions

Résumé:
Ce document contient les exercices du Module 05 des C++ modules.

Version: 9.1

Table des matières

Chapitre I

Introduction

C++ is a general-purpose programming language created by Bjarne Stroustrup as an extension of the C programming language, or "C with Classes" (source : Wikipedia).

C++ est un langage de programmation compilé permettant la programmation sous de multiples paradigmes, dont la programmation procédurale, la programmation orientée objet et la programmation générique. Ses bonnes performances, et sa compatibilité avec le C en font un des langages de programmation les plus utilisés dans les applications où la performance est critique (source : Wikipedia).

Ces modules ont pour but de vous introduire à la **Programmation Orientée Objet**. Plusieurs langages sont recommandés pour l'apprentissage de l'OOP. Du fait qu'il soit dérivé de votre bon vieil ami le C, nous avons choisi le langage C++. Toutefois, étant un langage complexe et afin de ne pas vous compliquer la tâche, vous vous conformerez au standard C++98.

Nous avons conscience que le C++ moderne est différent sur bien des aspects. Si vous souhaitez pousser votre maîtrise du C++, c'est à vous de creuser après le tronc commun de 42 !

Chapitre II

Consignes générales

Compilation

- Compilez votre code avec `c++` et les flags `-Wall -Wextra -Werror`
- Votre code doit compiler si vous ajoutez le flag `-std=c++98`

Format et conventions de nommage

- Les dossiers des exercices seront nommés ainsi : `ex00, ex01, ... , exn`
- Nommez vos fichiers, vos classes, vos fonctions, vos fonctions membres et vos attributs comme spécifié dans les consignes.
- Rédigez vos noms de classe au format **UpperCamelCase**. Les fichiers contenant le code d'une classe porteront le nom de cette dernière. Par exemple : `NomDeClasse.hpp`/`NomDeClasse.h`, `NomDeClasse.cpp`, ou `NomDeClasse.tpp`. Ainsi, si un fichier d'en-tête contient la définition d'une classe "BrickWall", son nom sera `BrickWall.hpp`.
- Sauf si spécifié autrement, tous les messages doivent être terminés par un retour à la ligne et être affichés sur la sortie standard.
- *Ciao Norminette !* Aucune norme n'est imposée durant les modules C++. Vous pouvez suivre le style de votre choix. Mais ayez à l'esprit qu'un code que vos pairs ne peuvent comprendre est un code que vos pairs ne peuvent évaluer. Faites donc de votre mieux pour produire un code propre et lisible.

Ce qui est autorisé et ce qui ne l'est pas

Le langage C, c'est fini pour l'instant. Voici l'heure de se mettre au C++ ! Par conséquent :

- Vous pouvez avoir recours à quasi l'ensemble de la bibliothèque standard. Donc plutôt que de rester en terrain connu, essayez d'utiliser le plus possible les versions C++ des fonctions C dont vous avec l'habitude.
- Cependant, vous ne pouvez avoir recours à aucune autre bibliothèque externe. Ce qui signifie que C++11 (et dérivés) et l'ensemble `Boost` sont interdits. Aussi, certaines fonctions demeurent interdites. Utiliser les fonctions suivantes résultera

en la note de 0 : `*printf()`, `*alloc()` et `free()`.

- Sauf si explicitement indiqué autrement, les mots-clés `using namespace <ns_name>` et `friend` sont interdits. Leur usage résultera en la note de -42.
- **Vous n'avez le droit à la STL que dans les Modules 08 et 09**. D'ici là, l'usage des **Containers** (vector/list/map/etc.) et des **Algorithmes** (tout ce qui requiert d'inclure `<algorithm>`) est interdit. Dans le cas contraire, vous obtiendrez la note de -42.

Quelques obligations côté conception

- Les fuites de mémoires existent aussi en C++. Quand vous allouez de la mémoire (en utilisant le mot-clé `new`), vous ne **devez pas avoir de memory leaks**.
- Du Module 02 au Module 09, vos classes devront se conformer à la forme **canonique, dite de Coplien, sauf si explicitement spécifié autrement**.
- Une fonction implémentée dans un fichier d'en-tête (hormis dans le cas de fonction template) équivaudra à la note de 0.
- Vous devez pouvoir utiliser vos fichiers d'en-tête séparément les uns des autres. C'est pourquoi ils devront inclure toutes les dépendances qui leur seront nécessaires. Cependant, vous devez éviter le problème de la double inclusion en les protégeant avec des **include guards**. Dans le cas contraire, votre note sera de 0.

Read me

- Si vous en avez le besoin, vous pouvez rendre des fichiers supplémentaires (par exemple pour séparer votre code en plus de fichiers). Vu que votre travail ne sera pas évalué par un programme, faites ce qui vous semble le mieux du moment que vous rendez les fichiers obligatoires.
- Les consignes d'un exercice peuvent avoir l'air simple mais les exemples contiennent parfois des indications supplémentaires qui ne sont pas explicitement demandées.
- Lisez entièrement chaque module avant de commencer ! Vraiment.
- Par Odin, par Thor ! Utilisez votre cervelle !!!

Vous aurez à implémenter un bon nombre de classes, ce qui pourrait s'avérer ardu... ou pas ! Il y a peut-être moyen de vous simplifier la vie grâce à votre éditeur de texte préféré.

Vous êtes assez libre quant à la manière de résoudre les exercices. Toutefois, respectez les consignes et ne vous en tenez pas au strict minimum, vous pourriez passer à côté de notions intéressantes. N'hésitez pas à lire un peu de théorie.

Chapitre III

Exercice 00 : Maman, quand je serai grand(e), je serai bureaucrate !

	Exercice : 00
Maman, quand je serai grand(e), je serai bureaucrate !	
Dossier de rendu : *ex*00/	
Fichiers à rendre : `Makefile, main.cpp, Bureaucrat.{h, hpp}, Bureaucrat.cpp`	
Fonctions interdites : `Aucune`	

`Notez que les classes d'exception n'ont pas à se conformer à la forme canonique de Coplien. Par contre, toutes les autres classes, si.`

Il est temps de créer un cauchemar artificiel de bureaux, de couloirs, de formulaires et d'heures perdues dans les files d'attente. Le rêve non ? Non ? Dommage.

Commençons par le plus petit rouage de cette vaste machine bureaucratique : le **Bureaucrat**.

Un **Bureaucrat** doit avoir :

- Un `name` (*nom*) constant
- Un `grade` (*échelon*) pouvant aller de **1** (échelon le plus élevé) à **150** (échelon le plus bas).

Toute tentative d'instancier un Bureaucrat en utilisant un échelon invalide jettera une exception :
`Bureaucrat::GradeTooHighException` ou `Bureaucrat::GradeTooLowException`.

Vous ajouterez des accesseurs pour chacun de ces attributs : `getName()` et `getGrade()`. Implémentez aussi deux fonctions membres afin d'incrémenter ou de décrémenter l'échelon du bureaucrate. Si ce dernier est trop haut ou trop bas, les mêmes exceptions que dans le constructeur seront jetées.

Rappelez-vous. Puisque l'échelon 1 est le plus élevé, et 150 le plus bas, incrémenter un échelon 3 donnera l'échelon 2 au bureaucrate.

Les exceptions jetées devront pouvoir être attrapées grâce à des blocs `try and catch` :

```
try
{
  /* do some stuff with bureaucrats */
}
catch (std::exception & e)
{
  /* handle exception */
}
```

Vous ajouterez également une surcharge de l'opérateur d'insertion («) afin d'afficher quelque chose comme (toujours sans les chevrons) :

`<name>, bureaucrat grade <grade>.`

Comme d'habitude, fournissez des tests afin de démontrer que tout marche comme attendu.

Chapitre IV

Exercice 01 : En rang, les larves !

	Exercice : 01
En rang, les larves !	
Dossier de rendu : *ex*01/	
Fichiers à rendre : `Fichiers des exercices précedents + Form.{h, hpp}, Form.cpp`	
Fonctions interdites : `Aucune`	

Maintenant que vous avez des bureaucrates, ce serait pas mal de leur donner quelque chose à faire. Quelle meilleure activité que celle de remplir une pile de formulaires ?

Par conséquent, il est temps de faire une classe **Form** (*formulaire*) qui possède :

- Un `name` (*nom*) constant.
- Un booléen indiquant si le formulaire est signé (à la construction, il ne l'est pas).
- Un `grade` (*échelon*) constant requis pour le signer.
- Un `grade` (*échelon*) constant requis pour l'exécuter.

Tous ces attributs sont **privés**, et non protégés.

Les échelons du **Form** suivent les mêmes règles que celles s'appliquant au Bureaucrat. Par conséquent, les exceptions suivantes seront jetées si un échelon est trop haut ou trop bas : `Form::GradeTooHighException` et `Form::GradeTooLowException`.

Comme précédemment, écrivez des accesseurs pour tous les attributs ainsi qu'une surcharge de l'opérateur d'insertion («) qui affiche toutes les informations du formulaire.

Ajoutez également au Form une fonction membre `beSigned()` prenant un Bureaucrat en paramètre. Il doit changer le status du formulaire en *signé* si l'échelon du Bureaucrat est suffisant (supérieur ou égal à l'échelon requis). Pour rappel, l'échelon 1 est plus élevé que l'échelon 2.
Si l'échelon est insuffisant, jetez une `Form::GradeTooLowException`.

Pour finir, ajoutez au Bureaucrat une fonction membre `signForm()`. Si le formulaire est signé, elle affichera quelque chose comme :

```
<bureaucrat> signed <form>
```

Ou dans le cas contraire :

```
<bureaucrat> couldn't sign <form> because <reason>.
```

Implémentez et rendez vos propres tests afin de démontrer que tout marche comme attendu.

Chapitre V

Exercice 02 : Ah mais non, il vous faut la 28B, pas la 28C...

	Exercice : 02
Ah mais non, il vous faut la 28B, pas la 28C...	
Dossier de rendu : *ex*02/	
Fichiers à rendre : `Makefile, main.cpp, Bureaucrat.[{h, hpp},cpp], Bureaucrat.cpp +` `AForm.[{h, hpp},cpp], ShrubberyCreationForm.[{h, hpp},cpp], + RobotomyRequestForm.[{h, hpp},cpp], PresidentialPardonForm.[{h, hpp},cpp]`	
Fonctions interdites : `Aucune`	

Étant donné que vous avez des formulaires de base, il est temps d'en créer d'autres qui auront une utilité.

Dans tous les cas, la classe de base Form doit être une classe abstraite et doit donc être renommée AForm. Gardez à l'esprit que les attributs du formulaire doivent rester privés et qu'ils se trouvent dans la classe de base. qu'ils se trouvent dans la classe de base.

Ajoutez les classes concrètes suivantes :

- **ShrubberyCreationForm** (*formulaire de création d'arbustes*) :
  Échelons requis : signature 145, exécution 137
  Créé un fichier `<target>_shrubbery` dans le répertoire courant, et écrit des arbres ASCII à l'intérieur.
- **RobotomyRequestForm** (*formulaire de demande de robotomie*) :
  Échelons requis : signature 72, exécution 45
  Fait des bruits de perceuse. Ensuite, informe que la `<target>` a été robotomisée avec succès 50% du temps. Dans le cas contraire, informe que l'opération a échoué.
- **PresidentialPardonForm** (*formulaire de pardon présidentiel*) :
  Échelons requis : signature 25, exécution 5
  Informe que la `<target>` a été pardonnée par Zaphod Beeblebrox.

Chacune d'entre elle prend un paramètre dans son constructeur : la `target` (*cible*) du

formulaire. Par exemple, "jardin" si vous voulez planter des arbustes dans votre jardin.

Maintenant, ajoutez la fonction membre `execute(Bureaucrat const & executor) const` à la classe de base et implémentez une fonction pour exécuter l'action du formulaire des classes dérivées. Vous devez vous assurer que le formulaire est signé et que le grade du bureaucrate tentant de l'exécuter est suffisant. Sinon, jetez une exception pertinente.

Que vous souhaitiez checker les prérequis dans chaque classe concrète ou dans la classe de base (puis appeler une autre fonction pour exécuter le formulaire), c'est votre choix. Toutefois, une de ces deux manières de faire est plus propre que l'autre.

Pour finir, ajoutez la fonction membre `executeForm(Form const & form)` au Bureaucrat. Ce dernier doit tenter d'exécute le formulaire. S'il y arrive, affichez un message comme :

```
<bureaucrat> executed <form>
```

Dans le cas contraire, affichez un message d'erreur explicite.

Implémentez et rendez vos propres tests afin de démontrer que tout marche comme attendu.

Chapitre VI

Exercice 03 : C'est toujours mieux que d'apporter le café

	Exercice : 03
C'est toujours mieux que d'apporter le café	
Dossier de rendu : *ex*03/	
Fichiers à rendre : `Fichiers des exercices précedents + Intern.{h, hpp}, Intern.cpp`	
Fonctions interdites : `Aucune`	

Parce que remplir des formulaires c'est pas fou, ce serait cruel de demander à nos bureaucrates de faire ça toute la journée. Heureusement qu'il y a les stagiaires. Dans cet exercice, vous implémenterez la classe **Intern** (*stagiaire*). Le stagiaire n'a pas de nom, pas d'échelon, aucun signe distinctif. La seule chose dont se préoccupent les bureaucrates, c'est qu'un stagiaire fasse son travail.

Cependant, le stagiaire a une aptitude importante : la fonction `makeForm()` qui prend deux strings en paramètres. La première est le nom du formulaire, la seconde la cible du formulaire. Elle retourne un pointeur sur un **objet Form** dont le nom est passé en paramètre et dont la cible est le second paramètre.

Elle affiche quelque chose comme :

```
Intern creates <form>
```

Si le nom du formulaire passé en paramètre n'existe pas, affichez un message d'erreur explicite.

Vous devez éviter les choses illisibles ou peu élégantes comme une forêt de if/elseif/else. Ce genre de solution ne sera pas accepté en évaluation. Vous n'êtes plus en Piscine. Comme d'habitude, testez que tout marche comme attendu.

Le code suivant crée un formulaire **RobotomyRequestForm** dont la cible est `"Bender"` :

```
{
    Intern  someRandomIntern;
    Form*   rrf;

    rrf = someRandomIntern.makeForm("robotomy request", "Bender");
}
```