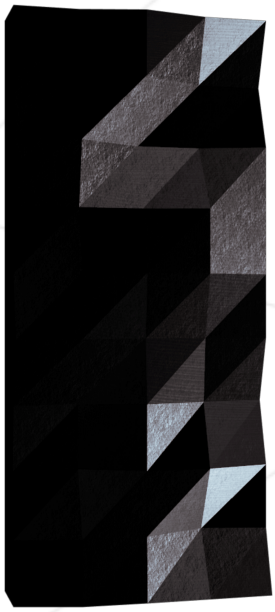

C++ - Module 04

Polymorphisme sous-type, classes abstraites, interfaces

Résumé: Ce document contient le sujet du module 04 des modules C++ de 42.

Table des matières

Chapitre I

Règles Générales

- Toute fonction déclarée dans une header (sans pour les templates) ou tout header non-protégé, signifie 0 à l'exercice.
- Tout output doit être affiché sur stdout et terminé par une newline, sauf autre chose est précisé.
- Les noms de fichiers imposés doivent être suivis à la lettre, tout comme les noms de classe, les noms de fonction, et les noms de méthodes.
- Rappel : vous codez maintenant en `C++`, et plus en `C`. C'est pourquoi :
  - Les fonctions suivantes sont **INTERDITES**, et leur usage se soldera par un 0 : `*alloc`, `*printf` et `free`
  - Vous avez l'autorisation d'utiliser à peu près toute la librairie standard. CEPENDANT, il serait intelligent d'essayer d'utiliser la version C++ de ce à quoi vous êtes habitués en C, plutôt que de vous reposer sur vos acquis. Et vous n'êtes pas autorisés à utiliser la STL jusqu'au moment où vous commencez à travailler dessus (module 08). Ca signifie pas de Vector/List/Map/etc... ou quoi que ce soit qui requiert une include <algorithm> jusque là.
- L'utilisation d'une fonction ou mécanique explicitement interdite sera sanctionnée par un `0`
- Notez également que sauf si la consigne l'autorise, les mot-clés `using namespace` et `friend` sont interdits. Leur utilisation sera punie d'un `0`.
- Les fichiers associés à une classe seront toujours nommés `ClassName.cpp` et `ClassName.hpp`, sauf si la consigne demande autre chose.
- Vous devez lire les exemples minutieusement. Ils peuvent contenir des prérequis qui ne sont pas précisés dans les consignes.
- Vous n'êtes pas autorisés à utiliser des librairies externes, incluant `C++11`, `Boost`, et tous les autres outils que votre ami super fort vous a recommandé
- Vous allez surement devoir rendre beaucoup de fichiers de classe, ce qui peut paraitre répétitif jusqu'à ce que vous appreniez a scripter ca dans votre éditeur de code préferé.

- Lisez complètement chaque exercice avant de le commencer.
- Le compilateur est `clang++`
- Votre code sera compilé avec les flags `-Wall -Wextra -Werror`
- Chaque include doit pouvoir être incluse indépendamment des autres includes. Un include doit donc inclure toutes ses dépendances.
- Il n'y a pas de norme à respecter en `C++`. Vous pouvez utiliser le style que vous préferez. Cependant, un code illisible est un code que l'on ne peut pas noter.
- Important : vous ne serez pas noté par un programme (sauf si précisé dans le sujet). Cela signifie que vous avez un degré de liberté dans votre méthode de résolution des exercices.
- Faites attention aux contraintes, et ne soyez pas fainéant, vous pourriez manquer beaucoup de ce que les exercices ont à offrir
- Ce n'est pas un problème si vous avez des fichiers additionnels. Vous pouvez choisir de séparer votre code dans plus de fichiers que ce qui est demandé, tant qu'il n'y a pas de moulinette.
- Même si un sujet est court, cela vaut la peine de passer un peu de temps dessus afin d'être sûr que vous comprenez bien ce qui est attendu de vous, et que vous l'avez bien fait de la meilleure manière possible.

Chapitre II

Exercice 00 : Polymorphisme, ou "Quand le sorcier pensait que vous soyiez plus mignon en mouton"

	Exercice : 00
Polymorphisme, ou "Quand le sorcier pensait que vous soyiez plus mignon en mouton"	
Dossier de rendu : $ex00/$	
Fichiers à rendre : `Sorcerer.hpp, Sorcerer.cpp, Victim.hpp, Victim.cpp, Peon.hpp, Peon.cpp, main.cpp`	
Fonctions interdites : `Aucune`	

Le polymorphisme est une tradition antique remontant à l'époque des mages, des sorciers et autres charlatans. Nous pourrions essayer de vous faire croire que nous y avons pensé d'abord, mais c'est un mensonge !

Intéressons-nous à notre ami Ro/b/ert, le Magnifique, sorcier de métier.

Robert a un passe-temps intéressant : transformer tout ce qu'il peut attraper : moutons, poneys, loutres, et beaucoup d'autres choses improbables (Jamais vu un gryphon ... ?).

Commençons par créer une classe Sorcerer, qui a un nom et un titre. Il a un constructeur prenant son nom et son titre comme paramètres (dans cet ordre).

Il ne peut pas être instancié sans paramètres (Cela n'aurait aucun sens ! Imaginez un sorcier sans nom ni titre ... Pauvre garçon, il ne pourrait pas se vanter devant les filles de la taverne ...). Mais vous devez toujours utiliser la forme Coplienne. Encore une fois, oui, il y a une certaine astuce impliquée.

A la naissance du sorcier, vous afficherez :

```
NOM, TITRE, is born!
```

(Bien entendu, vous devez remplacer NOM et TITRE avec le nom du sorcier et son titre, respectivement).

À sa mort, vous afficherez :

```
NOM, TITRE, is dead. Consequences will never be the same!
```

Un sorcier doit pouvoir s'introduire proprement :

```
I am NOM, TITRE, and i like ponies!
```

Il peut s'introduire sur n'importe quel output, grâce à un overload de l'opérateur «. (Rappel : l'utilisation de friend est interdite. Ajoutez les getters nécessaires).

Notre sorcier a maintenant besoin de victimes, pour s'amuser le matin, entre les griffes d'ours et le jus de troll.

Créez alors une classe `Victime`. Un peu comme le sorcier, il aura un nom, et un constructeur qui prend son nom en paramètre.

À la naissance de la victime, affichez :

```
A random victim called NOM just appeared!
```

À sa mort, affichez :

```
The victim NOM died for no apparent reasons!
```

La victime peut aussi se présenter, dans la même veine que le Sorcier, et dire :

```
Je suis NOM and I like otters!
```

Notre `Victim` peut être "polymorphed()" par le `Sorcerer`. Ajoutez une méthode `void getPolymorphed() const` qui dira :

```
NOM was just polymorphed in a cute little sheep!
```

Ajoutez également la fonction membre `void polymorph(Victim const &) const` à votre `Sorcerer`, afin de polymorpher les gens.

Maintenant, pour ajouter un peu de variété, notre `Sorcerer` aimerait polymorpher d'autre choses, pas seulement une `Victim` générique.
Aucun problème ! nous allons en créer d'autres !

Faites une classe `Peon`.

`Un Peon est une Victim. Donc...`

À sa naissance, il doit dire "Zog zog.", et à sa mort, "Bleuark..." (Regardez l'exemple, c'est pas aussi simple que prévu).
Le `Peon` doit être polymorphé de la manière suivante :

```
NOM was just polymorphed into a pink pony!
```

(Un poNymorphe, AHAHAHA.. Ahaha.. ah)

Le code qui suit doit compiler, et afficher l'output correct :

```
int main()
{
	Sorcerer robert("Robert", "the Magnificent");

	Victim jim("Jimmy");
	Peon joe("Joe");

	std::cout << robert << jim << joe;

	robert.polymorph(jim);
	robert.polymorph(joe);

	return 0;
}
```

Output :

```
$> clang++ -W -Wall -Werror *.cpp
$> ./a.out | cat -e
Robert, the Magnificent, is born!$
A random victim called Jimmy just appeared!$
A random victim called Joe just appeared!$
Zog zog.$
I am Robert, the Magnificent, and i like ponies!$
Je suis Jimmy and I like otters!$
Je suis Joe and I like otters!$
Jimmy was just polymorphed in a cute little sheep!$
Joe was just polymorphed in a cute little sheep!$
Bleuark...$
Victim Joe died for no apparent reasons!$
Victim Jimmy died for no apparent reasons!$
Robert, the Magnificent, is dead. Consequences will never be the same!$
$>
```

Si vous êtes vraiment consciencieux vous pourriez faire plus de tests : ajout de classes dérivées, etc... (Faites-le !)

N'oubliez pas de rendre votre propre `main`, parce que tout ce qui n'est pas testé ne sera pas noté.

Chapitre III

Exercice 01 : Je ne veux pas brûler le monde

	Exercice : 01
Je ne veux pas brûler le monde	
Dossier de rendu : *ex*01/	
Fichiers à rendre : `AWeapon.[hpp,cpp]`, `PlasmaRifle.[hpp,cpp]`, `PowerFist.[hpp,cpp]`, `Enemy.[hpp,cpp]`, `SuperMutant.[hpp,cpp]`, `RadScorpion.[hpp,cpp]`, `Character.[hpp,cpp]`, `main.cpp`	
Fonctions interdites : `Aucune`	

Dans les Wasteland, vous pouvez trouver beaucoup de choses. Des morceaux de métal, des produits chimiques étranges, des croisements entre cow-boys et punks sans-abri, mais aussi une cargaison d'armes improbables (mais drôles !). Et il est temps aussi, je voulais casser des gueules aujourd'hui.

Pour que nous puissions survivre dans toute cette bordel, vous allez commencer par nous coder des armes. Complétez et implémentez la classe suivante (n'oubliez pas la forme Coplienne...) :

```
class AWeapon
{
        private:
                [...]

        public:
                AWeapon(std::string const & name, int apcost, int damage);
                [...] ~AWeapon();
                std::string [...] getName() const;
                int getAPCost() const;
                int getDamage() const;
                [...] void attack() const = 0;
};
```

Info :

- Une arme a un nom, une valeur de dégâts, et un coût en AP (Action Points) pour tirer.
- Une arme a un bruit et des effets visuels associés quand vous utilisez attack() avec. Ceci sera géré dans les classes héritées.

Implémentez ensuite la classe concrète `PlasmaRifle` et `PowerFist`.
Voici leurs caractéristiques :

- PlasmaRifle :
  - Name : "Plasma Rifle"
  - Damage : 21
  - AP cost : 5
  - Output of attack() : "* piouuu piouuu piouuu *"
- PowerFist :
  - Name : "Power Fist"
  - Damage : 50
  - AP cost : 8
  - Output of attack() : "* pschhh... SBAM ! *"

Et voilà ! Maintenant que nous avons plein d'armes pour nous battre, nous allons rajouter des ennemis à affronter (écraser, défoncer, dézingeur, clouer aux portes, désintégrer, etc...).

Faites une classe `Enemy`, avec le modèle suivant (Bien entendu, vous devez faire une classe Coplienne) :

```
class Enemy
{
        private:
                [...]

        public:
                Enemy(int hp, std::string const & type);
                [...] ~Enemy();
                std::string [...] getType() const;
                int getHP() const;

                virtual void takeDamage(int);
};
```

Contraintes :

- Un ennemi a un nombre de points de vie et un type.
- Un ennemi peut prendre des dégâts (qui réduisent ses HP). Si les dégâts sont $<$ 0, ne faites rien.

Vous allez implémenter quelques ennemis concrets. Qu'on puisse s'amuser.

Premièrement, la classe `SuperMutant`. Gros, méchant, moche, et avec le QI d'une plante en pot. C'est un peu comme un éléphant dans un couloir. Si vous le ratez, vous

faites exprès. Somme toute un très bon punchingball pour vous entraîner.

Voici ces caractéristiques :

- HP : 170
- Type : "Super Mutant"
- À la naissance, affichez : "Gaaah. Break everything !"
- À la mort, affichez : "Aaargh ..."
- Overloadez `takeDamage` pour réduire en permanence les dégâts subis de 3. (Ils sont forts comme ca !)

Ensuite, faites une classe `RadScorpion`. C'est pas une bête SI sauvage, mais quand même : Un scorpion géant a un charme certain non ?

- Caractéristiques :
  - HP : 80
  - Type : "RadScorpion"
  - À la naissance, affichez : "* click click click *"
  - À la mort, affichez : "* SPROTCH *"

Maintenant, nous avons des armes, des ennemis pour les tester, plus qu'à créer le héros !

Vous allez donc créer la classe `Character`, avec le modèle suivant (vous connaissez la chanson) :

```
class Character
{
        private:
                [...]

        public:
                Character(std::string const & name);
                [...]
                ~Character();
                void recoverAP();
                void equip(AWeapon*);
                void attack(Enemy*);
                std::string [...] getName() const;
};
```

- Il a un nom, un nombre de points d'action, et un pointeur vers `AWeapon`, qui représente l'arme actuellement équipée.
- À sa création, votre héros a 40 AP, et en perds à chaque utilisation de l'arme. Il récupère 10 AP à chaque appel à `recoverAP()` avec un maximum de 40 AP. Si vous n'avez pas d'AP, vous ne pouvez pas attaquer.
- Affiche "NOM attaque ENEMY_TYPE with a WEAPON_NAME" après un appel à `attack()`, suivi par un appel `attack()` de l'arme actuelle. Si aucune arme n'est

équippée, `attack()` ne fait rien. Vous retirerez les HP à l'ennemi basé sur les dégâts de l'arme. Si les hp de l'ennemi passent à 0, vous devez le détruire.

- `equip()` doit stocker uniquement un pointeur sur l'arme, il n'y a pas de copie.

Vous devez également implémenter un overload de l'opérateur « pour afficher les caractéristiques de votre personnage. Ajoutez tous les getters nécessaires.

Cet overload affichera :

```
NAME has AP_NUMBER AP and carries a WEAPON_NAME
```

Si il a une arme. Sinon, affichez :

```
NOM has AP_NUMBER AP and is unarmed
```

Voici une petite fonction main (basique) pour tester. La votre doit être meilleure :

```
int main()
{
	Character* moi = new Character("moi");

	std::cout << *moi;

	Enemy* b = new RadScorpion();

	AWeapon* pr = new PlasmaRifle();
	AWeapon* pf = new PowerFist();

	moi->equip(pr);
	std::cout << *moi;
	moi->equip(pf);

	moi->attack(b);
	std::cout << *moi;
	moi->equip(pr);
	std::cout << *moi;
	moi->attack(b);
	std::cout << *moi;
	moi->attack(b);
	std::cout << *moi;

	return 0;
}
```

Output :

```
$> clang++ -W -Wall -Werror *.cpp
$> ./a.out | cat -e
moi has 40 AP and is unarmed$
* click click click *$
moi has 40 AP and carries a Plasma Rifle$
moi hasttaque RadScorpion with a Power Fist$
* pschhh... SBAM! *$
moi has 32 AP and carries a Power Fist$
moi has 32 AP and carries a Plasma Rifle$
moi hasttaque RadScorpion with a Plasma Rifle$
* piouuu piouuu piouuu *$
moi has 27 AP and carries a Plasma Rifle$
moi hasttaque RadScorpion with a Plasma Rifle$
* piouuu piouuu piouuu *$
* SPROTCH *$
moi has 22 AP and carries a Plasma Rifle$
```

Comme d'habitude, rendez un `main` qui inclue vos tests.

Chapitre IV

Exercice 02 : This code is unclean. Purify it !

	Exercice : 02
This code is unclean. Purify it !	
Dossier de rendu : *ex*02/	
Fichiers à rendre : `Squad.hpp, Squad.cpp, TacticalMarine.hpp, TacticalMarine.cpp, AssaultTerminator.hpp, AssaultTerminator.cpp, ISpaceMarine.hpp, ISquad.hpp, main.cpp`	
Fonctions interdites : `Aucune`	

Votre mission est de construire une armée dont l'apparence n'a d'égal que sa violence.

Vous allez devoir implémenter une classe `Squad` et une classe `TacticalMarine` (pou remplir votre escouade).

Commençons avec la `Squad`. Voicei l'interface que vous allez devoir implémenter (incluez `ISquad.hpp`) :

```
class ISquad
{
       public:
              virtual ~ISquad() {}
              virtual int getCount() const = 0;
              virtual ISpaceMarine* getUnit(int) const = 0;
              virtual int push(ISpaceMarine*) = 0;
};
```

Vous devez l'implémenter de telle sorte que :

- `getCount()` renvoie le nombre d'unités actuellement dans l'escouade.
- `getUnit(N)` renvoie un pointeur vers l'unité N (commencant par 0)
- `push(XXX)` ajoute l'unité XXX au bout de la `Squad` (Ajouter une unité NULL ou une unité deja dans une Squad n'a bien entendu aucun sens).

Finalement, la classe `Squad` que nous vous demandons est un simple container pour vos Marines, que nous utiliserons pour structurer correctement notre armée.

L'assignation par copie d'une `Squad` doit impliquer une `deep copy`. Si il y avait une unité dans l'escouade avant, elle doit être détruite avant d'être remplacée. Chaque unité sera créée avec `new()`.

Lorsque qu'une `Squad` est détruite, chaque unité au sein de l'escouade est détruite aussi, dans l'ordre.

Voici l'interface à respecter pour implémenter les `TacticalMarine`.
`ISpaceMarine.hpp` :

```
class ISpaceMarine
{
       public:
              virtual ~ISpaceMarine() {}
              virtual ISpaceMarine* clone() const = 0;
              virtual void battleCry() const = 0;
              virtual void rangedAttack() const = 0;
              virtual void meleeAttack() const = 0;
};
```

Contraintes :

- `clone()` renvoie une copie de l'object actuel.
- À la création, affiche : "Tactical Marine ready for action !"
- `battleCry()` affiche "For the Holy PLOT !"
- `rangedAttack()` affiche "* attacks with a bolter *"
- `meleeAttack()` affiche "* attacks with a chainsword *"
- À la mort, affiche : "Aaargh ..."

De la même manière, implémentez un `AssaultTerminator`, avec les affichages suivants :

- Naissance : "* teleports from space *"
- `battleCry()` : "This code is unclean. Purify it !"
- `rangedAttack` : "* does nothing *"
- `meleeAttack` : "* attaque with chainfists *"
- Mort : "I'll be back ..."

Here's a bit of test code. As usual, yours should be more thorough.

```
int main()
{
```

```
        ISpaceMarine* bob = new TacticalMarine;
        ISpaceMarine* jim = new AssaultTerminator;

        ISquad* vlc = new Squad;
        vlc->push(bob);
        vlc->push(jim);
        for (int i = 0; i < vlc->getCount(); ++i)
        {
                ISpaceMarine* cur = vlc->getUnit(i);
                cur->battleCry();
                cur->rangedAttack();
                cur->meleeAttack();
        }
        delete vlc;

        return 0;
}
```

Output :

```
$> clang++ -W -Wall -Werror *.cpp
$> ./a.out | cat -e
Tactical Marine ready for action!$
* teleports from space *$
For the Holy PLOT!$
* attacks with a bolter *$
* attaque with chainsword *$
This code is unclean. Purify it!$
* does nothing *$
* attaque with chainfists *$
Aaargh ...$
I'll be back ...$
```

Soyez consciencieux avec votre `main` si vous voulez une bonne note.

Chapitre V

Exercice 03 : Bocal Fantasy

	Exercice : 03
Bocal Fantasy	
Dossier de rendu : *ex*03/	
Fichiers à rendre : `AMateria.hpp, AMateria.cpp, Ice.hpp, Ice.cpp, Cure.hpp, Cure.cpp, Character.hpp, Character.cpp, MateriaSource.hpp, MateriaSource.cpp, ICharacter.hpp, IMateriaSource.hpp, main.cpp`	
Fonctions interdites : `Aucune`	

`Cet exercice et ceux qui suivent ne rapportent pas de points, mais demeurent interessant dans le cadre de votre piscine. Vous n'êtes pas obligés de les faire.`

Complétez la définition de la clase `AMateria`, et implémentez les fonctions membres nécessaires.

```
class AMateria
	{
	private:
		[...]
		unsigned int _xp;

	public:
		AMateria(std::string const & type);
		[...]
		[...] ~AMateria();

		std::string const & getType() const; //Returns the materia type
		unsigned int getXP() const; //Returns the Materia's XP

		virtual AMateria* clone() const = 0;
```

```
virtual void use(ICharacter& target);
```

Le système d'expérience d'une Materia fonctionne comme suit :

- l'XP totale d'une Materia demarre de 0, et augmente de 10 à chaque appel à `use()`. Trouvez une manière intelligente de gérer ça.

Créer les classes concrètes `Ice` et `Cure`. Leur type sera le nom de la classe en minuscule. Leur méthode `clone()` devra, bien entendu, renvoyer une nouvelle instance du vrai type de Materia.
La méthode `use(ICharacter&)` affichera :

- Ice : "* shoots an ice bolt at NOM *"
- Cure : "* heals NOM's wounds *"

(Vous remplacerez bien entendu NOM par le `Character` donné en paramètre).

Lorsque vous assigner une Materia à une autre, copier le type n'a pas de sens.

Créez la classe `Character`, qui implémentera l'interface suivante :

```
class ICharacter
{
       public:
              virtual ~ICharacter() {}
              virtual std::string const & getName() const = 0;
              virtual void equip(AMateria* m) = 0;
              virtual void unequip(int idx) = 0;
              virtual void use(int idx, ICharacter& target) = 0;
};
```

Le Character a un inventaire qui contient jusqu'à 4 Materia. Il démarre vide.
Il équipera les Materia dans l'ordre, de 0 à 3.
Si l'inventaire et plein et que l'on essaye d'équiper une Materia, ou utiliser/dé-équiper une Materia non-existante, ne faites rien. `unequip()` ne doit PAS supprimer une Materia. `use(int, ICharacter&)` devra utiliser la Materia dans le slot dont l'index est donné, sur la cible donnée. Il passera la cible à la méthode `AMateria::use`.

Bien entendu, vous devez accepter n'importe quel type de AMateria dans votre inventaire.

Votre `Character` doit avoir un constructeur qui prend son nom en paramètre. L'assignation par copie doit être une `deepcopy`, bien entendu. L'ancienne `Materia` du `Character`

doit être supprimée. De même lors de la destruction d'un `Character`.

Maintenant que votre personnage peut équiper et uiliser de la Materia, ca commence à ressembler à quelque chose.

Ceci dit, nous n'avons pas envie de créer manuellement cette Materia. Crééz une classe `MateriaSource`, qui implémentera l'interface suivante :

```
class IMateriaSource
{
	public:
		virtual ~IMateriaSource() {}
		virtual void learnMateria(AMateria*) = 0;
		virtual AMateria* createMateria(std::string const & type) = 0;
};
```

`learnMateria` doit copier la Materia passée en paramètre, et la stocker en mémoire pour pouvoir la cloner plus tard. De la même manière que `Character`, la Source peut connaitre au maximum 4 Materia, qui ne sont pas obligatoirement uniques.

`createMateria(std::string const &)` renvoie une nouvelle Materia, qui sera une copie de la Materia (apprise précedemment) dont le type équivaut au paramètre. Renvoyez 0 si le type est inconnu.

En conclusion, votre Source doit être capable d'apprendre des "templates" de Materia et les re-créer sur demande. Vous allez pouvoir être capable de créer des Materia sans connaitre leur veritable type, juste en utilisant une string qui l'identifie.

Voilà le main initial, sur lequel vous devez travailler :

```
int main()
{
    IMateriaSource* src = new MateriaSource();
    src->learnMateria(new Ice());
    src->learnMateria(new Cure());

    ICharacter* moi = new Character("moi");

    AMateria* tmp;
    tmp = src->createMateria("ice");
    moi->equip(tmp);
    tmp = src->createMateria("cure");
    moi->equip(tmp);

    ICharacter* bob = new Character("bob");

    moi->use(0, *bob);
    moi->use(1, *bob);

    delete bob;
    delete moi;
    delete src;

    return 0;
}
```

Output :

```
$> clang++ -W -Wall -Werror *.cpp
$> ./a.out | cat -e
* shoots an ice bolt at bob *$
* heals bob's wounds *$
```

N'oubliez pas de rendre votre propre `main`.

Chapitre VI

Exercice 04 : AFK Mining

	Exercice : 04
AFK Mining	
Dossier de rendu : $ex04/$	
Fichiers à rendre : `DeepCoreMiner.[hpp,cpp]`, `StripMiner.[hpp,cpp]`, `AsteroKreog.[hpp,cpp]`, `KoalaSteroid.[hpp,cpp]`, `MiningBarge.[hpp,cpp]`, `IAsteroid.hpp`, `IMiningLaser.hpp`, `main.cpp`	
Fonctions interdites : `typeid() ou autre, lisez les warnings`	

`Pour cet exercice, l'utilisation de typeid() est complètement interdite.`

À première vue, vous pourriez penser que l'espace au-delà de `Kreog` n'est qu'un vaste néant. Mais non, bon monsieur, en fait il abrite un quantité incroyable de choses aléatoires et inutiles.

Entre bimbos de l'espace, des monstres hideux, des corbeilles spatiales et même ces affreux développeurs kernel, vous y trouverez une quantité colossale d'astéroïdes, tous remplis de minéraux plus précieux les uns que les autres. Un peu comme la ruée vers l'or, juste sans Picsou.

Vous voilà, prospecteur d'espace. Vous venez d'arriver. Pour éviter de ressembler à un Débutant complet, vous aurez besoin d'outils. Et comme les pioches sont pour les débutants, nous utiliserons des lasers.

Voici l'interface à implémenter pour vos lasers de minage :

```
class IMiningLaser
{
       public:
              virtual ~IMiningLaser() {}
              virtual void mine(IAsteroid*) = 0;
};
```

Implémentez les deux classes suivantes : `DeepCoreMiner` et `StripMiner`.
Leur `mine(IAsteroid*)` donnera les résultats suivants :

- `DeepCoreMiner`

```
"* mining deep ... got RESULT ! *"
```

- `StripMiner`

```
"* strip mining ... got RESULT ! *"
```

Vous remplacerez `RESULT` avec le retour de `beMined` qui arrive de l'astéroide ciblé.

Nous allons du coup avoir besoin d'astéroides, pour, euh, miner. Voici l'interface :

```
class IAsteroid
{
       public:
              virtual ~IAsteroid() {}
              virtual std::string beMined([...] *) const = 0;
              [...]
              virtual std::string getName() const = 0;
};
```

Les deux astéroides à implémenter sont l'`Asteroid` et la `Comet`. Leur méthode `getName()` renverra leur nom (évidemment), qui sera le nom de la classe.

En utilisant le sous-type et le polymorphisme paramétrique (et votre cerveau), vous devez faire en sorte qu'un call à `IMiningLaser::mine` renvoie un résultat différent selon le type de laser le type d'astéroide.

Les retours seront tels que suit :

- `StripMiner` et `Comet` : "Tartarite"

- `DeepCoreMiner` et `Comet` : "Mithril"
- `StripMiner` et `Asteroid` : "Flavium"
- `DeepCoreMiner` et `Asteroid` : "Dragonite"

Pour atteindre ce résultat, vous devez compléter l'interface `IAsteroid`.

`Vous allez probablement avoir besoin de deux méthodes beMined...`

`N'essayez pas de déduire le retour de l'astéroide via getName(). Vous DEVEZ utiliser les TYPES et le POLYMORPHISME. Toute autre approche (typeid, dynamic_cast, getName, etc...) est interdite.`

C'est pas aussi dur que cela en a l'air. Maintenant que nos jouets sont enfin prêts, faites-vous une petite barge sympathique pour aller miner. Implémenter la classe qui suit :

```
class MiningBarge
{
	public:
		void equip(IMiningLaser*);
		void mine(IAsteroid*) const;
};
```

- Une barge commence sans laser, et peut en équiper jusqu'à 4.
- Si elle a déjà 4 lasers, `equip(IMiningLaser*)` ne fait rien.
- La méthode `mine(IAsteroid*)` appelle `IMiningLaser::mine` de tous les lasers équippés, dans l'ordre où ils ont été équippés.

Good luck.