



## Piscine C++ - d10

Non, les Koalas ne prennent pas de drogue. Poney.

Maxime “zaz” Montinet [zaz@epitech.eu](mailto:zaz@epitech.eu)

*Abstract: Ce document est le sujet du d10*

# Table des matières

I	REMARQUES GÉNÉRALES	2
II	Exercice 0	4
III	Exercice 1	8
IV	Exercice 2	13
V	Exercice 3	17
VI	Exercice 4	21

# Chapitre I

## REMARQUES GÉNÉRALES


- LISEZ LES REMARQUES GÉNÉRALES ATTENTIVEMENT!!!!
  - Vous n'aurez aucune excuse si vous avez 0 parce que vous avez oublié une consigne générale ...
- REMARQUES GÉNÉRALES :
  - Si vous faites la moitié des exercices car vous avez du mal, c'est normal. Par contre, si vous faites la moitié des exercices par flemme et vous tirez à 14h, vous AUREZ des surprises. Ne tentez pas le diable.
  - Toute fonction implémentée dans un header ou header non protégé signifie 0 à l'exercice.
  - Toutes les classes doivent posséder un constructeur et un destructeur.
  - Toutes les sorties se font sur la sortie standard et sont terminées par un retour à la ligne sauf si le contraire est précisé explicitement.
  - Les noms de fichiers qui vous sont imposés doivent être respectés À LA LETTRE, de même que les noms de classes et de fonctions membres / méthodes.
  - Rappelez-vous que vous faites du C++ et non plus du C. Par conséquent, les fonctions suivantes sont INTERDITES, et leur utilisation sera sanctionnée par un -42 :
    - `*alloc`
    - `*printf`
    - `free`
    - `open`, `fopen`, etc ...
  - De façon générale, les fichiers associés à une classe seront `NOM_DE_LA_CLASSE.hh`

et `NOM_DE_LA_CLASSE.cpp` (s'il y a lieu).

- Les répertoires de rendus sont `ex00`, `ex01`, ..., `exN`
  - Toute utilisation de `friend` se soldera par un -42, **no questions asked** .
  - Lisez attentivement les exemples, ils peuvent requérir des choses que le sujet ne dit pas...
  - Ces exercices vous demandent de rendre beaucoup de classes, mais la plupart sont TRÈS courtes si vous faites ça intelligemment. Donc, halte à la flemme !
  - Lisez ENTièrement le sujet d'un exercice avant de le commencer !
  - REFLÉCHISSEZ. Par pitié.
  - REFLÉCHISSEZ. Par Odin !
  - R.E.F.L.É.C.H.I.S.S.E.Z. Nom d'une pipe.
- COMPILATION DES EXERCICES :
    - La moulinette compile votre code avec les flags : `-W -Wall -Werror`
    - Pour éviter les problèmes de compilation de la moulinette, incluez les fichiers nécessaires dans vos fichiers `include (*.hh)`.
    - Notez bien qu'aucun de vos fichiers ne doit contenir de fonction `main` . Nous utiliserons notre propre fonction `main` pour compiler et tester votre code.
    - Ce sujet peut être modifié jusqu'à 4h avant le rendu. Rafraichissez-le régulièrement !
    - Le répertoire de rendu est : `(DÉPOT SVN - piscine_cpp_d10-promo-login_x)/exX` (N étant bien sur le numéro de l'exercice).

# Chapitre II

## Exercice 0

	Exercice : 00	points : 3
Le polymorphisme c'est quand le sorcier trouve que tu serais vachement plus mignon en mouton.		
Répertoire de rendu: (DÉPOT SVN - piscine_cpp_d10-promo-login_x)/ex00		
Compilateur : g++	Flags de compilation: -W -Wall -Werror	
Makefile : Non	Règles : n/a	
Fichiers a rendre : Sorcerer.hh, Sorcerer.cpp, Victim.hh, Victim.cpp, Peon.hh, Peon.cpp		
Remarques : n/a		
Fonctions Interdites : Aucune		

Le polymorphisme, c'est une tradition séculaire qui remonte au temps des mages, sorciers et autres charlatans. On a beau vous faire croire qu'on y a pensé les premiers, c'est faux !

Intéressons nous a notre ami Ro/b/ert, le Magnifique, sorcier de son état.

Robert a un passe-temps vachement marrant, c'est de changer tout ce qui lui passe sous le bras en mouton, en poney, en loutre, et autres choses diverses, variées voire improbables (Vous avez déjà vu un périfalque ... ?).

Commençons par créer une classe Sorcerer, possédant un nom et un titre. Il dispose d'un constructeur prenant son nom et son titre en paramètres (dans cet ordre).

Il ne peut pas être instancié sans paramètres. (Ca n'aurait pas de sens ! Imaginez un sorcier sans nom ou sans titre. Le pauvre, il ne pourrait pas se faire mousser devant les ribaudes à la taverne.)

A la naissance d'un sorcier, on affichera :

```
1 NAME, TITLE, is born !
```

(Bien entendu, remplacez NAME et TITLE par le nom et le titre, respectivement...)

De même, à sa mort, on affichera :

```
1 NAME, TITLE, is dead. Consequences will never be the same !
```

Un Sorcerer doit pouvoir se présenter de la manière suivante :

```
1 I am NAME, TITLE, and I like ponies !
```

Il se présentera sur un flux de sortie au choix, grace à une surcharge d'opérateur « sur ostream, comme vous savez si bien en faire.

(Rappel : Il est interdit d'utiliser friend pour cette journée. Rajoutez tous les getters dont vous avez besoin !)

Notre sorcier a besoin de victimes, pour s'amuser un peu le matin entre les croissants et le jus de troll.

Vous allez donc créer une classe `Victim` . Un peu comme le sorcier, elle possède un nom, et a un constructeur prenant son nom en paramètre.

À la naissance d'une victime, affichez :

```
1 Some random victim called NAME just popped !
```

À sa mort, affichez :

```
1 Victim NAME just died for no apparent reason !
```

La Victim se présentera également, de la même façon que le Sorcerer (c'est à dire avec une surcharge d'opérateur « sur ostream), et dira :

```
1 I'm NAME and i like otters !
```

Notre `Victim` pourra se faire "polymorpher" par le `Sorcerer` . Ajoutez donc une méthode `void getPolymorphed() const` à votre `Victim` , qui lui fera dire :

```
1 NAME has been turned into a cute little sheep !
```

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

Maintenant, pour un peu plus de variété, notre `Sorcerer` voudrait polymorpher d'autres choses qu'une `Victim` générique. Qu'à cela ne tienne, vous allez créer d'autres victimes...

Créez une classe `Peon` .



Un `Peon` EST-UN Victim. Donc...

À sa naissance, il affichera “Zog zog.”, et a sa mort “Bleuark...” (Tip : Regardez bien l'exemple. Ce n'est pas si simple.) Le `Peon` se fera polymorpher de la façon suivante :

```
1 NAME has been turned into a pink pony !
```

(C'est un peu un poNymorph ...)

Le code suivant devra compiler et afficher la sortie ci-apres :

```
1 int main()
2 {
3     Sorcerer robert("Robert", "the Magnificent");
4
5     Victim jim("Jimmy");
6     Peon joe("Joe");
7
8     std::cout << robert << jim << joe;
9
10    robert.polymorph(jim);
11    robert.polymorph(joe);
12
13    return 0;
14 }
```


Sortie :

```
1 zaz@blackjack ex00 $ g++ -W -Wall -Werror *.cpp
2 zaz@blackjack ex00 $ ./a.out | cat -e
3 Robert, the Magnificent, is born !$
4 Some random victim called Jimmy just popped !$
5 Some random victim called Joe just popped !$
6 Zog zog.$
7 I am Robert, the Magnificent, and I like ponies !$
8 I'm Jimmy and i like otters !$
9 I'm Joe and i like otters !$
10 Jimmy has been turned into a cute little sheep !$
11 Joe has been turned into a pink pony !$
12 Bleuark...$
13 Victim Joe just died for no apparent reason !$
14 Victim Jimmy just died for no apparent reason !$
15 Robert, the Magnificent, is dead. Consequences will never be the same !$
16 zaz@blackjack ex00 $
```



# Chapitre III

## Exercice 1

	Exercice : 01	points : 3
I don't want to set the world on fire...		
Répertoire de rendu: (DÉPOT SVN - piscine_cpp_d10-promo-login_x)/ex01		
Compilateur : g++	Flags de compilation: -W -Werror -Wall	
Makefile : Non	Règles : n/a	
Fichiers a rendre : AWeapon.[hh,cpp], PlasmaRifle.[hh,cpp], PowerFist.[hh,cpp], AEnemy.[hh,cpp], SuperMutant.[hh,cpp], RadScorpion.[hh,cpp], Character.[hh,cpp]		
Remarques : n/a		
Fonctions Interdites : Aucune		

Dans le Wasteland, on trouve un sacré paquet de trucs. Des bouts de ferraille, des produits bizarres, des croisements de cowboys et de SDF à tendance punk, mais aussi plein d'armes festives et rigolotes.

Tant mieux, j'avais envie de frapper des trucs aujourd'hui.

Histoire de survivre un peu dans tout ce merdier, vous allez donc commencer par coder des armes. Complétez et implémentez la classe suivante :

```
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;
};
```

Explications :

- Une arme a un nom, un nombre de points de dégâts infligés, et un coût en AP (points d'action) par tir.
- L'arme fait certains effets sons et lumières quand on `attack()` avec. Cela sera laissé aux sous-classes.

Implémentez ensuite les classes concrètes `PlasmaRifle` et `PowerFist`. Voici leurs caractéristiques :

- `PlasmaRifle` :
  - Nom : "Plasma Rifle"
  - Dégâts : 21
  - Coût AP : 5
  - Sortie de `attack()` : "\* piouuu piouuu piouuu \*"
- `PowerFist` :
  - Nom : "Power Fist"
  - Dégâts : 50
  - Coût AP : 8
  - Sortie de `attack()` : "\* pschhh... SBAM! \*"

Voilà ... Maintenant qu'on a de jolies armes pour faire joujou, il va falloir des ennemis à combattre! (Ou défoncer, clouer aux portes, kreogiser, merger leur rectum avec leur tête, bref ...)

Faites nous donc une classe `AEnemy` sur le modèle suivant (à compléter bien sur) :

```
1 class AEnemy
2 {
3     private:
4         [...]
5
6     public:
7         AEnemy(int hp, std::string const & type);
8         [...] ~AEnemy();
9         std::string [...] getType() const;
10        int getHP() const;
11
12        virtual void takeDamage(int);
13};
```

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

Vous allez ensuite implémenter quelques ennemis, histoire de se défouler.

D'abord, le **SuperMutant**. Grand, gros, moche, et un QI d'ordinaire plutôt associé à un pot de fleurs en terre cuite qu'à un organisme vivant. Ceci dit, c'est un peu comme un **Mancubus** dans un couloir : c'est difficile à rater. C'est donc un excellent punching-ball pour vous faire la main.

Voici ses caractéristiques :

- HP : 170
- Type : "Super Mutant"
- Affiche à la naissance : "Gaaah. Me want smash heads!"
- Affiche à la mort : "Aaargh ..."
- Surcharge **takeDamage** pour prendre 3 points de dégâts en moins (Eh oui, ils ont la peau dure, ces enflures.)

Ensuite, faites nous un **RadScorpion**. Pas bien farouche, ceci dit un scorpion géant a toujours un certain cachet, vous ne trouvez pas ?

- Caractéristiques :
  - HP : 80
  - Type : "RadScorpion"
  - Affiche à la naissance : "\* click click click \*"
  - Affiche à la mort : "\* SPROTCH \*"

Nous avons maintenant des armes, et des ennemis pour les essayer. Parfait. Il ne nous reste plus qu'à exister nous-mêmes.

Créez donc la classe **Character**, sur le modèle suivant :

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

    public:
```

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

- Explications :

- Possède un nom, un nombre de AP (Points d'action), et un pointeur sur **AWeapon** pour représenter l'arme courante.
- Possède 40 AP au départ, perd les AP correspondants à l'arme quand il en utilise une, et récupère 10 AP à chaque appel à **recoverAP()** , pour un maximum de 40. Si pas assez d'AP, pas d'attaque.
- Affiche "NAME attacks ENEMY\_TYPE with a WEAPON\_NAME" lors d'un appel à **attack()** , suivi ensuite d'un appel à la methode **attack()** de l'arme courante. S'il n'y a pas d'arme équipée, **attack()** n'a aucun effet. On enlève ensuite des HP a l'ennemi selon les dégats de l'arme. Après cela, si la cible a 0 HP ou moins, vous devez delete la cible.
- **equip()** se contente de stocker un pointeur sur l'arme, il n'y a pas de copie d'objet a faire.

Vous ferez également une surcharge de l'operateur « sur ostream pour afficher les attributs de votre **Character** . Ajoutez y tous les getters nécessaires. Cette surcharge affichera :

```
1 NAME has AP_NUMBER AP and wields a WEAPON_NAME
```

si une arme est equipee, sinon :

```
1 NAME has AP_NUMBER AP and is unarmed
```

Voici un main de test basique :


```
1 int main()
2 {
3     Character* zaz = new Character('zaz');
4
5     std::cout << *zaz;
6
7     AEnemy* b = new RadScorpion();
8
9     AWeapon* pr = new PlasmaRifle();
10    AWeapon* pf = new PowerFist();
11
12    zaz->equip(pr);
13    std::cout << *zaz;
14    zaz->equip(pf);
15
16    zaz->attack(b);
17    std::cout << *zaz;
18    zaz->equip(pr);
19    std::cout << *zaz;
20    zaz->attack(b);
21    std::cout << *zaz;
22    zaz->attack(b);
23    std::cout << *zaz;
24
25    return 0;
26 }
```

Sortie :

```
1 zaz@blackjack ex01 $ g++ -W -Wall -Werror *.cpp
2 zaz@blackjack ex01 $ ./a.out | cat -e
3 zaz has 40 AP and is unarmed$
4 * click click click *$
5 zaz has 40 AP and wields a Plasma Rifle$
6 zaz attacks RadScorpion with a Power Fist$
7 * pschhh... SBAM! *$
8 zaz has 32 AP and wields a Power Fist$
9 zaz has 32 AP and wields a Plasma Rifle$
10 zaz attacks RadScorpion with a Plasma Rifle$
11 * piouuu piouuu piouuu *$
12 zaz has 27 AP and wields a Plasma Rifle$
13 zaz attacks RadScorpion with a Plasma Rifle$
14 * piouuu piouuu piouuu *$
15 * SPROTCH *$
16 zaz has 22 AP and wields a Plasma Rifle$
```

# Chapitre IV

## Exercice 2

	Exercice : 02	points : 4
This code is unclean. PURIFY IT!		
Répertoire de rendu: (DÉPOT SVN - piscine_cpp_d10-promo-login_x)/ex02		
Compilateur : g++	Flags de compilation: -W -Werror -Wall	
Makefile : Non	Règles : n/a	
Fichiers a rendre : Squad.hh, Squad.cpp, TacticalMarine.hh, TacticalMarine.cpp, AssaultTerminator.hh, AssaultTerminator.cpp		
Remarques : n/a		
Fonctions Interdites : Aucune		

Votre mission est de construire une armée digne des Valiant Lion Crusaders. Peinte avec des rayures blanches et orange. Si, si, je vous jure.

Vous devez implémenter les éléments de base de votre future armée, à savoir une escouade ( **Squad** ) et un Space Marine tactique ( **TacticalMarine** ).

Commençons par **Squad** . Voici l'interface à implémenter (Include **ISquad.hh** ):

```

1 class ISquad
2 {
3     public:
4         virtual ~ISquad() {}
5         virtual int getCount() const = 0;
6         virtual ISpaceMarine* getUnit(int) = 0;
7         virtual int push(ISpaceMarine*) = 0;
8 };

```

Votre implémentation devra faire en sorte que :

- **getCount()** renvoie le nombre d'unités actuellement dans l'escouade.
- **getUnit(N)** renvoie un pointeur vers l'unité N (Bien entendu, on commence à 0. Pointeur nul si index hors limites.)
- **push(XXX)** ajoute l'unité XXX à la fin de l'escouade. Renvoie le nombre d'unités

dans l'escouade après l'opération. (Ajouter une unité 'nulle', ou déjà dans l'escouade, n'a bien entendu pas de sens...)

Au bout du compte, la **Squad** que l'on vous demande est un simple conteneur de Space Marines, afin d'organiser correctement votre armée.

Lors d'une construction par copie ou d'une assignation de **Squad**, la copie doit être profonde. Lors d'une assignation, s'il y avait des unités avant, elles sont détruites avant d'être remplacées. Partez du principe que toutes les unités seront créées avec **new**. Quand une **Squad** est détruite, les unités qui en font partie le sont aussi, dans l'ordre.

Pour **TacticalMarine**, voici l'interface à implémenter (Include **ISpaceMarine.hh**):

```
1 class ISpaceMarine
2 {
3     public:
4         virtual ~ISpaceMarine() {}
5         virtual ISpaceMarine* clone() const = 0;
6         virtual void battleCry() const = 0;
7         virtual void rangedAttack() const = 0;
8         virtual void meleeAttack() const = 0;
9 };
```

Contraintes:

- **clone()** renvoie une copie de l'objet courant
- À la naissance, affiche "Tactical Marine ready for battle"
- **battleCry()** affiche "For the holy PLOT!"
- **rangedAttack** affiche "\* attacks with bolter \*"
- **meleeAttack** affiche "\* attacks with chainsword \*"
- À la mort, affiche "Aaargh ..."

De la même façon, implémentez aussi un **AssaultTerminator**, avec les sorties suivantes:

- Naissance : "\* teleports from space \*"
- **battleCry()** : "This code is unclean. PURIFY IT!"
- **rangedAttack** : "\* does nothing \*"
- **meleeAttack** : "\* attacks with chainfists \*"
- Mort : "I'll be back ..."



Veillez noter que vos classes doivent respecter PARFAITEMENT les interfaces. Nous testerons vos implémentations de concert avec les nôtres !



Voici un code de test :


```
1 int main()
2 {
3     ISpaceMarine* bob = new TacticalMarine;
4     ISpaceMarine* jim = new AssaultTerminator;
5
6     ISquad* vlc = new Squad;
7     vlc->push(bob);
8     vlc->push(jim);
9     for (int i = 0; i < vlc->getCount(); ++i)
10    {
11        ISpaceMarine* cur = vlc->getUnit(i);
12        cur->battleCry();
13        cur->rangedAttack();
14        cur->meleeAttack();
15    }
16    delete vlc;
17
18    return 0;
19 }
```

Sortie :

```
1 zaz@blackjack ex02 $ g++ -W -Wall -Werror *.cpp
2 zaz@blackjack ex02 $ ./a.out | cat -e
3 Tactical Marine ready for battle$
4 * teleports from space *$
5 For the holy PLOT !$
6 * attacks with bolter *$
7 * attacks with chainsword *$
8 This code is unclean. PURIFY IT !$
9 * does nothing *$
10 * attacks with chainfists *$
11 Aaargh ...$
12 I'll be back ...$
```

# Chapitre V

## Exercice 3

	Exercice : 03	points : 5
Kreog Fantasy VII		
Répertoire de rendu: (DÉPOT SVN - piscine_cpp_d10-promo-login_x)/ex03		
Compilateur : g++	Flags de compilation: -W -Werror -Wall	
Makefile : Non	Règles : n/a	
Fichiers a rendre : AMateria.hh, AMateria.cpp, Ice.hh, Ice.cpp, Cure.hh, Cure.cpp, Character.hh, Character.cpp, MateriaSource.hh, MateriaSource.cpp		
Remarques : n/a		
Fonctions Interdites : Aucune		

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

```

1 class AMateria
2 {
3     private:
4         [...]
5         unsigned int xp_;
6
7     public:
8         AMateria(std::string const & type);
9         [...]
10        [...] ~AMateria();
11
12        std::string const & getType() const; //Donne le type de la
13        materia
14        unsigned int getXP() const; //Retourne l'XP de la materia
15
16        virtual AMateria* clone() const = 0;
17        virtual void use(ICharacter& target);
18 };

```

Le système d'XP d'une `Materia` fonctionne comme suit:

La `Materia` a un capital d'XP qui démarre à 0, et qui augmente de 10 à chaque appel à `use()`. À vous de trouver une façon intelligente de gérer cela !

Créez les `Materia` concrètes `Ice` et `Cure`. Leur type devra être leur nom en minuscule ("ice" pour `Ice`, "cure" pour `Cure`).

Leur méthode `clone` devra, évidemment, renvoyer une nouvelle instance du type réel de la `Materia`.

En ce qui concerne la méthode `use(ICharacter&)`, elle affichera :

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

(Bien évidemment, remplacer `NAME` par le nom du `Character` passe en paramètre).



Bien entendu que lors de l'assignation d'une `AMateria` à une autre, copier le type n'a pas de sens...

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

```
1 class ICharacter
2 {
3     public:
4         virtual ~ICharacter() {}
5         virtual std::string const & getName() const = 0;
6         virtual void equip(AMateria* m) = 0;
7         virtual void unequip(int idx) = 0;
8         virtual void use(int idx, ICharacter& target) = 0;
9 };
```

Le `Character` possède un inventaire de 4 `Materia` au maximum, au départ son inventaire est bien entendu vide. Il équipera les `Materia` dans les slots 0 à 3, dans l'ordre. Dans le cas où on essaierait d'équiper une `Materia` avec l'inventaire plein, ou de `use/unequip` une `Materia` qui n'est pas là, ne faites rien.

La méthode `unequip` ne doit pas delete les `Materia` !

La méthode `use(int, ICharacter&)` devra utiliser la `Materia` à l'emplacement `idx` et passer en paramètre `target` à la méthode `AMateria::use`.



Bien entendu, vous devrez pouvoir supporter n'importe quelle `AMateria` dans l'inventaire de vos `Character`.

Votre `Character` doit avoir un constructeur prenant son nom en paramètre. La copie ou assignation d'un `Character` doit être profonde, évidemment. Les anciennes `Materia` d'un `Character` doivent être delete. Pareil lors de la destruction du `Character`.

Maintenant que vos personnages peuvent équiper et utiliser des `Materia`, ça commence à ressembler à quelque chose.

Ceci étant dit, c'est embêtant de devoir créer nos `Materia` à la main, et de devoir connaître leur type ...

Vous allez donc créer une Source intelligente de `Materia`.

Créez la classe `MateriaSource`, qui devra implémenter l'interface suivante :

```
1 class IMateriaSource
2 {
3     public:
4         virtual ~IMateriaSource() {}
5         virtual void learnMateria(AMateria*) = 0;
6         virtual AMateria* createMateria(std::string const & type) = 0;
7 };
```

`learnMateria` doit copier la `Materia` passée en paramètre, et la garder en mémoire afin d'être reproduite plus tard. De la même façon que le `Character`, la `Source` peut connaître au maximum 4 `Materia`, qui ne sont pas nécessairement uniques.

`createMateria(std::string const &)` va renvoyer une nouvelle `Materia` qui sera une copie de la `Materia` (préalablement apprise par la `Source`) dont le type correspond au paramètre. Renvoiera 0 si le type est inconnu.

En bref, votre `Source` doit pouvoir apprendre des "modèles" de `Materia`, et les recréer à la demande.

Vous allez ainsi pouvoir créer une `Materia` sans avoir à connaître son type "réel", juste une chaîne de caractères qui identifie son type.

Elle est pas belle, la vie ?


```
1 int main()
2 {
3     IMateriaSource* src = new MateriaSource();
4     src->learnMateria(new Ice());
5     src->learnMateria(new Cure());
6
7     ICharacter* zaz = new Character("zaz");
8
9     AMateria* tmp;
10    tmp = src->createMateria("ice");
11    zaz->equip(tmp);
12    tmp = src->createMateria("cure");
13    zaz->equip(tmp);
14
15    ICharacter* bob = new Character("bob");
16
17    zaz->use(0, *bob);
18    zaz->use(1, *bob);
19
20    delete bob;
21    delete zaz;
22    delete src;
23
24    return 0;
25 }
```

Sortie :

```
1 zaz@blackjack ex03 $ g++ -W -Wall -Werror *.cpp
2 zaz@blackjack ex03 $ ./a.out | cat -e
3 * shoots an ice bolt at bob *$
4 * heals bob's wounds *$
```

# Chapitre VI

## Exercice 4

	Exercice : 04	points : 6
Ce soir, je serai la plus barge pour aller miner !		
Répertoire de rendu: (DÉPOT SVN - piscine_cpp_d10-promo-login_x)/ex04		
Compilateur : g++	Flags de compilation: -W -Werror -Wall	
Makefile : Non	Règles : n/a	
Fichiers a rendre : DeepCoreMiner.[hh,cpp] , StripMiner.[hh,cpp] , AsteroKreog.[hh,cpp] , KoalaSteroid.[hh,cpp] , MiningBarge.[hh,cpp] , IAsteroid.hh		
Remarques : n/a		
Fonctions Interdites : Pour cet exercice, l'utilisation de typeid() est RIGOREUSEMENT INTERDITE et assimilée à de la TRICHE. Tout typeid dans votre code résultera en un -42, SANS EXCEPTION. Vous êtes prévenus.		

On croirait peut-être, à première vue, que l'espace derrière la **KoalaGate** est une grande étendue de vide. Mais il n'en est rien, en fait c'est rempli de bordel divers et varié.

Entre des Space Bimbos, des monstres tentaculaires et ignobles, des détritrus de l'espace, et même parfois des développeurs Microsoft, on y trouve une quantité phénoménale d'astéroïdes, tous remplis de minerais plus précieux les uns que les autres. Un peu comme la ruée vers l'or, les cowboys en moins.

Vous voici donc bombardé prospecteur spatial. Pour ne pas passer pour un complet pignouf, il va vous falloir des outils. Et comme les pioches c'est fait pour les faibles et les hommes de peu, nous on utilise des lasers.

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

```
1 class IMiningLaser
2 {
3     public:
4         virtual ~IMiningLaser() {}
5         virtual void mine(IAsteroid*) = 0;
6 };
```

Implémentez les deux lasers concrets `DeepCoreMiner` et `StripMiner`.

Leur méthode `mine(IAsteroid*)` affichera la sortie indiquée :

- `DeepCoreMiner`

```
1      '* mining deep ... got RESULT ! *'
```

- `StripMiner`

```
1      '* strip mining ... got RESULT ! *'
```

Vous remplacerez `RESULT` par le retour de `beMined` de l'astéroïde cible.

Il nous faut aussi des astéroïdes à pomp... euh, miner. Voici leur interface :

```
1 class IAsteroid
2 {
3     public:
4         virtual ~IAsteroid() {}
5         virtual std::string beMined(...) const = 0;
6         [...]
7         virtual std::string getName() const = 0;
8 };
```

Les deux astéroïdes à implémenter sont `AsteroKreog` et `KoalaSteroid`. Leur méthode `getName` renverra bien évidemment leur nom, qui sera égal au nom de la classe.

Vous devez, en utilisant de façon intelligente les polymorphismes par héritage et paramétrique, faire en sorte que lors d'un appel à `IMiningLaser::mine`, le résultat soit fonction du type de l'astéroïde ET du type du laser.

Les retours sont comme suit :

- `StripMiner` sur `KoalaSteroid` : "Koalite"
- `DeepCoreMiner` sur `KoalaSteroid` : "Zazium"
- `StripMiner` sur `AsteroKreog` : "Kreogium"
- `DeepCoreMiner` sur `AsteroKreog` : "Sullite"

Vous devez, à cette fin, compléter l'interface `IAsteroid` .



Il vous faudra probablement deux méthodes `beMined` ... Elles prendraient leur paramètre par pointeur non `const`, et seraient `const` toutes les deux... Ne rajoutez pas autre chose. (Sinon la moulinette vous détruira avec vos propres tripes.)



N'essayez pas de faire des retours en fonction du `getName` de l'asteroide. Vous DEVEZ vous servir des TYPES et des polymorphismes. Toute façon détournée (`typeid`, `dynamic_cast`, `getName`, etc ...) vous vaudra un -42. (Oui, même si vous êtes un petit malin. La moulinette vous aura, faites moi confiance).

Réfléchissez, ce n'est pas si difficile que ça.



Spatcheur ! Spatcheur ! (Copyright 2010 "La blague a Zaz !")

Maintenant que vos jouets sont fin prêts, construisez vous une belle barge pour aller miner. Implémentez la classe suivante:

```
1 class MiningBarge
2 {
3     public:
4         void equip(IMiningLaser*);
5         void mine(IAsteroid*) const;
6 };
```

- Une barge démarre sans aucun laser et peut en équiper 4, pas un de plus. Si elle a déjà 4 lasers, `equip(IMiningLaser*)` ne fait rien. (Note : on ne copie pas...)
- La méthode `mine(IAsteroid*)` fait appel à `IMiningLaser::mine` de tous les lasers équipés dans l'ordre où ils ont été équipés.

Bon courage.

PS : Non, vous n'aurez pas de main de test. Vous êtes grands maintenant.