


# Chapter VII

## Exercise 03: Bocal Fantasy

	Exercise 03
Bocal Fantasy	
Turn-in directory : <i>ex03/</i>	
Files to turn in : <i>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</i>	
Forbidden functions : None	
Remarks : n/a	

Complete the definition of the following `AMateria` class, and implement the necessary member functions.

```
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);
};
```

A Materia's XP system works as follows:

A `Materia` has an XP total starting at 0, and increasing by 10 upon every call to `use()`. Find a smart way to handle that !

Create the concrete `Materias` `Ice` and `Cure`. Their type will be their name in lowercase ("ice" for Ice, etc ...).

Their `clone()` method will, of course, return a new instance of the real `Materia`'s type.

Regarding the `use(ICharacter&)` method, it'll display:

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

(Of course, replace NAME by the name of the `Character` given as parameter.)



While assigning a `Materia` to another, copying the type doesn't make sense...

Create the `Character` class, which will implement the following interface :

```
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;
};
```

The `Character` possesses an inventory of 4 `Materia` at most, empty at start. He'll equip the `Materia` in slots 0 to 3, in this order.

In case we try to equip a `Materia` in a full inventory, or use/unequip a nonexistent `Materia`, don't do a thing.

The `unequip` method must NOT delete `Materia` !

The `use(int, ICharacter&)` method will have to use the `Materia` at the `idx` slot, and pass `target` as parameter to the `AMateria::use` method.



Of course, you'll have to be able to support ANY `AMateria` in a `Character`'s inventory.

Your `Character` must have a constructor taking its name as parameter. Copy or assignation of a `Character` must be deep, of course. The old `Materia` of a `Character` must be deleted. Same upon destruction of a `Character`.

Now that your characters can equip and use `Materia`, it's starting to look right.

That being said, I would hate to have to create `Materia` by hand, and therefore have to know its real type...  
So, you'll have to create a smart Source of `Materia`.

Create the `MateriaSource` class, which will have to implement the following interface :

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

`learnMateria` must copy the `Materia` passed as parameter, and store it in memory to be cloned later. Much in the same way as for `Character`, the Source can know at most 4 `Materia`, which are not necessarily unique.

`createMateria(std::string const &)` will return a new `Materia`, which will be a copy of the `Materia` (previously learned by the Source) which type equals the parameter. Returns 0 if the type is unknown.

In a nutshell, your Source must be able to learn “templates” of `Materia`, and re-create them on demand. You'll then be able to create a `Materia` without knowing its “real” type, just a string identifying it. Life's good, eh ?

As usual, here's a test main that you'll have to improve on :

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

    ICharacter* zaz = new Character("`zaz'");

    AMateria* tmp;
    tmp = src->createMateria("`ice'");
    zaz->equip(tmp);
    tmp = src->createMateria("`cure'");
    zaz->equip(tmp);

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

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

    delete bob;
    delete zaz;
    delete src;

    return 0;
}
```


Output :

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

Don't forget to turn in your main function, because you... well, okay, you know the drill now, don't you ?

# Chapter VIII

## Exercise 04: AFK Mining

	Exercise 04
AFK Mining	
Turn-in directory : <i>ex04/</i>	
Files to turn in : <code>DeepCoreMiner.[hpp,cpp]</code> , <code>StripMiner.[hpp,cpp]</code> , <code>AsteroKreog.[hpp,cpp]</code> , <code>KoalaSteroid.[hpp,cpp]</code> , <code>MiningBarge.[hpp,cpp]</code> , <code>IAsteroid.hpp</code> , <code>IMiningLaser.hpp</code> , <code>main.cpp</code>	
Forbidden functions : <code>typeid()</code> and more, read the warnings	
Remarks : n/a	



For this exercise, the use of `typeid()` is absolutely FORBIDDEN and would result in a -42 to the day. That would be bad.

On first sight, you might think that the space beyond the `KreogGate` is just vast nothingness. But no, good sir, actually it's home to a metric fuckton of random useless stuff.

Between Space Bimbos, hideous monsters, space trash and even some filthy web developers, you'll find a colossal quantity of asteroids there, all filled with minerals each more precious than the last. A little bit like the goldrush, just without Scrooge McDuck.

Here you are, freshly started space prospector. To avoid looking like a complete redneck, you're gonna need some tools. And since pickaxes are for the lesser men, we use lasers.

Here's the interface to implement for your mining lasers :

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

Implement the two following concrete lasers : `DeepCoreMiner` and `StripMiner` .

Their `mine(IAsteroid*)` method will give the following output :

- `DeepCoreMiner`

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

- `StripMiner`

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

You'll replace `RESULT` with the return of `beMined` from the target asteroid.

We'll also need some asteroids to pum... er, i mean mine. Here's the corresponding interface :

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

The two asteroids to implement are the `AsteroBocal` and the `BocalSteroid` . Their `getName()` method will return their name (You don't say ?), which will be equal to the class name.

Using subtype and parametric polymorphisms (and your brain, hopefully), you will do so that a call to `IMiningLaster::mine` yields a result depending on the type of

asteroid AND the type of laser.

The returns will be as follows :

- StripMiner on BocalSteroid : "Krpite"
- DeepCoreMiner on BocalSteroid : "Zazium"
- StripMiner on AsteroBocal : "Flavium"
- DeepCoreMiner on AsteroBocal : "Thorite"

To that end, you will need to complete the `IAsteroid` interface.



You probably will need two `beMined` methods ... They would take their parameter by non-const pointer, and would both be `const`.



Don't try to deduce the return from the asteroid's `getName()`. You NEED to use TYPES and POLYMORPHISMS. Any other devious way (`typeid`, `dynamic_cast`, `getName`, etc ...) WILL net you a -42. (Yes, even if you think you can get away with it. Because no, you can't.)

Think. It's not that hard.



DD's patcher. (Copyright 2010 "zaz's daily joke")

Now that our toys are finally ready, make yourself a nice barge to go mine with. Implement the following class :

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

- A barge starts without a laser, and can equip 4 of them, not more. If it already has 4 lasers, `equip(IMiningLaser*)` does nothing. (Hint: We don't copy.)

- The `mine(IAsteroid*)` method calls `IMiningLaser::mine` from all the equipped lasers, in the order they were equipped in.

Good luck.

PS : No, you won't have any test main function. You're big boys now, make your own.

<insert a witty comment about how the students need to turn in their `main` function to get a grade, preferably with some veiled insults on Microsoft developers>