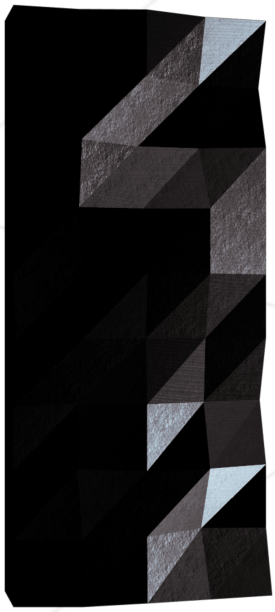

C++ - Module 04

Subtype polymorphism, abstract classes, interfaces

Summary: This document contains the subject for Module 04 of the C++ modules.

Contents

Chapter I

General rules

- Any function implemented in a header (except in the case of templates), and any unprotected header, means 0 to the exercise.
- Every output goes to the standard output, and will be ended by a newline, unless specified otherwise.
- The imposed filenames must be followed to the letter, as well as class names, function names and method names.
- Remember: You are coding in `C++` now, not in `C` anymore. Therefore:
  - The following functions are FORBIDDEN, and their use will be punished by a `0`, no questions asked: `*alloc`, `*printf` and `free`.
  - You are allowed to use basically everything in the standard library. HOWEVER, it would be smart to try and use the C++-ish versions of the functions you are used to in C, instead of just keeping to what you know, this is a new language after all. And NO, you are not allowed to use the STL until you actually are supposed to (that is, until module 08). That means no vectors/lists/maps/etc... or anything that requires an include <algorithm> until then.
- Actually, the use of any explicitly forbidden function or mechanic will be punished by a `0`, no questions asked.
- Also note that unless otherwise stated, the `C++` keywords `"using namespace"` and `"friend"` are forbidden. Their use will be punished by a `-42`, no questions asked.
- Files associated with a class will always be `ClassName.hpp` and `ClassName.cpp`, unless specified otherwise.
- Turn-in directories are `ex00/`, `ex01/`, ..., `exn/`.
- You must read the examples thoroughly. They can contain requirements that are not obvious in the exercise's description. If something seems ambiguous, you don't understand `C++` enough.
- Since you are allowed to use the `C++` tools you learned about since the beginning, you are not allowed to use any external library. And before you ask, that also means

no `C++11` and derivates, nor `Boost` or anything your awesomely skilled friend told you `C++` can't exist without.

- You may be required to turn in an important number of classes. This can seem tedious, unless you're able to script your favorite text editor.
- Read each exercise FULLY before starting it! Really, do it.
- The compiler to use is `clang++`.
- Your code has to be compiled with the following flags : `-Wall -Wextra -Werror`.
- Each of your includes must be able to be included independently from others. Includes must contains every other includes they are depending on, obviously.
- In case you're wondering, no coding style is enforced during in `C++`. You can use any style you like, no restrictions. But remember that a code your peer-evaluator can't read is a code she or he can't grade.
- Important stuff now : You will NOT be graded by a program, unless explictly stated in the subject. Therefore, you are afforded a certain amount of freedom in how you choose to do the exercises. However, be mindful of the constraints of each exercise, and DO NOT be lazy, you would miss a LOT of what they have to offer !
- It's not a problem to have some extraneous files in what you turn in, you may choose to separate your code in more files than what's asked of you. Feel free, as long as the result is not graded by a program.
- Even if the subject of an exercise is short, it's worth spending some time on it to be absolutely sure you understand what's expected of you, and that you did it in the best possible way.
- By Odin, by Thor! Use your brain!!!

Chapter II

Exercise 00: Polymorphism, or "When the sorcerer thinks you'd be cuter as a sheep"

	Exercise : 00
Polymorphism, or "When the sorcerer thinks you'd be cuter as a sheep"	
Turn-in directory : $ex00/$	
Files to turn in : `Sorcerer.hpp, Sorcerer.cpp, Victim.hpp, Victim.cpp, Peon.hpp, Peon.cpp, main.cpp, plus the needed files for your tests`	
Forbidden functions : `None`	

Polymorphism is an antic tradition, dating back to the time of mages, sorcerers, and other charlatans. We might try to make you think we thought of it first, but that's a lie!

Let's take an interest in our friend Ro/b/ert, the Magnificent, sorcerer by trade.

Robert has an interesting pastime: Morphing everything he can lay his hands on into sheeps, ponies, otters, and many other improbable things (Ever seen a perifalk...?).

Let's begin by creating a Sorcerer class, which has a name and a title. It has a constructor taking the sorcerers name and title as parameters (in this order).

The class can't be instanciated without parameters (That wouldn't make any sense! Imagine a sorcerer with no name, or no title... Poor guy, he couldn't boast to the wenches at the tavern...). But you still have to use Coplien's form. Again, yes, there is some form of trick involved. We're shifty like that.

At the birth of a sorcerer, you will display:

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

(Of course, you will replace NAME and TITLE with the sorcerer's name and title, respectively.)

At his death, you will display:

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

A sorcerer has to be able to introduce himself thusly:

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

He can introduce himself on any output stream, thanks to an overload of the << to ostream operator (you know how to do it!).
(Reminder: the use of friend is forbidden. Add every getter you need!)

Our sorcerer now needs victims, to amuse himself in the morning, between bear claws and troll juice.

Therefore you will create a `Victim` class. A little like the sorcerer, it will have a name, and a constructor taking its name as parameter.

At the birth of a victim, display:

```
Some random victim called NAME just appeared!
```

At its death, display:

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

The victim can also introduce itself, in the very same way as the Sorcerer, and says:

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

Our `Victim` can be "polymorphed" by the `Sorcerer`. Add a `void getPolymorphed() const` method to the `Victim`, which will say:

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

Also add the `void polymorph(Victim const &) const` member function to your `Sorcerer`, so you can polymorph people.

Now, to add a little variety, our `Sorcerer` would like to polymorph something else, not only a generic `Victim` . Not a problem, you'll just create some more!

Make a `Peon` class.

`A Peon is a Victim. So...`

At birth, he will say "Zog zog.", and at his death, "Bleuark..." (Tip: Watch the example. It's not that simple...) The `Peon` will get polymorphed thusly:

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

(It's kind of a poNymorph...)

The following code must compile, and display the following output:

```
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
$> ./a.out | cat -e
Robert, the Magnificent, is born!$
Some random victim called Jimmy just appeared!$
Some random victim called Joe just appeared!$
Zog zog.$
I am Robert, the Magnificent, and I like ponies!$
I'm Jimmy and I like otters!$
I'm Joe and I like otters!$
Jimmy has been turned into a cute little sheep!$
Joe has been turned into a pink pony!$
Bleuark...$
Victim Joe just died for no apparent reason!$
Victim Jimmy just died for no apparent reason!$
Robert, the Magnificent, is dead. Consequences will never be the same!$
$>
```

If you're really thorough, you could make some more tests: Add derived classes, etc... (No, it's not actually a suggestion, you really should do it.)

Of course, as usual, you will turn in your `main` function, because anything that's not tested will not be graded.

Chapter III

Exercise 01: I don't want to set the world on fire

	Exercise : 01
I don't want to set the world on fire	
Turn-in directory : $ex01/$	
Files to turn in : `AWeapon.[hpp,cpp], PlasmaRifle.[hpp,cpp], PowerFist.[hpp,cpp], Enemy.[hpp,cpp], SuperMutant.[hpp,cpp], RadScorpion.[hpp,cpp], Character.[hpp,cpp], main.cpp, plus the needed files for your tests`	
Forbidden functions : `None`	

In the Wasteland, you can find a great many things. Bits of metal, strange chemicals, crosses between cowboys and homeless wannabe punks, but also a boatload of improbable (but funny!) weapons. And it's about time too, I wanted to hit some stuff in the face today.

Just so we can survive in all this crap, you're going to start by coding us some weapons. Complete and implement the following class (Don't forget Coplien's form...):

```
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:

- A weapon has a name, a number of damage points inflicted upon a hit, and a shooting cost in AP (action points).
- A weapon produces certain sounds and lighting effects when you attack() with it. This will be deferred to the inheriting classes.

After that, you can implement the concrete classes `PlasmaRifle` and `PowerFist` . Here are their characteristics:

- 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! *"

There we go. Now that we have plenty of shiny weapons to play with, we're gonna need some enemies to fight! (Or disperse, piledrive, nail to doors, kreogize, etc...)

Make an `Enemy` class, with the following model (You'll have to complete it, obviously, and again, Coplien...):

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

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

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

Constraints:

- An enemy has a number of hit points and a type.
- An enemy can take damage (which reduces his HP). If the damage is <0, don't do anything.

You'll then implement some concrete enemies. Just to have fun with.

First, the `SuperMutant` . Big, bad, ugly, and with an IQ ordinarily associated more with a flowerpot than a living being. That being said, it's a bit like a `Mancubus` in a hallway: if you miss him, you're really doing it on purpose. So, it's an excellent punching-ball to train yourself with.

Here are its characteristics:

- HP: 170
- Type: "Super Mutant"
- On birth, displays: "Gaaah. Me want smash heads!"
- Upon death, displays: "Aaargh..."
- Overloads `takeDamage` to take 3 less damage points than normal (Yeah, they're kinda strong, these guys.)

Then, make us a `RadScorpion` . Not that savage of a beast, I'll admit. But still, a giant scorpion does have a certain something to it, right?

- Characteristics:
  - HP: 80
  - Type: "RadScorpion"
  - On birth, displays: "* click click click *"
  - Upon death, displays: "* SPROTCH *"

Now that we have weapons, and enemies to try them on, we just need to exist ourselves.

So, you're going to create the `Character` class, with the following model (you know the drill):

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

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

- Has a name, a number of AP (Action points), and a pointer to `AWeapon` representing the current weapon.

- Posesses 40 AP at creation, loses the AP corresponding to the weapon he has on each use, and recovers 10 AP upon each call to `recoverAP()` , up to a maximum of 40. No AP, no attack.
- Displays "NAME attacks ENEMY_TYPE with a WEAPON_NAME" upon a call to `attack()` , followd by a call to the current weapon's `attack()` method. If there's no equipped weapon, `attack()` doesn't do a thing. You'll then substract to the enemy's HP the damage value of the weapon. After that, if the target has 0 HP, you must delete it.
- equip() will just store a pointer to the weapon, there's no copy involved.

You will also implement an overload of the << to ostream operator to display the attributes of your `Character` . Add every necessary getter function.

This overload will display:

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

if there's a weapon equipped. Else, it will display:

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

Here's a (pretty basic) test main function. Yours should be better.

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

	std::cout << *me;

	Enemy* b = new RadScorpion();

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

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

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

	return 0;
}
```

Output:

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

Have some fun creating more weapons and different enemies and turn in a `main` with

your tests.

Chapter IV

Exercise 02: This code is unclean. PURIFY IT!

	Exercise : 02
This code is unclean. PURIFY IT!	
Turn-in directory : $ex02/$	
Files to turn in : `Squad.hpp, Squad.cpp, TacticalMarine.hpp, TacticalMarine.cpp, AssaultTerminator.hpp, AssaultTerminator.cpp, ISpaceMarine.hpp, ISquad.hpp, main.cpp`	
Forbidden functions : `None`	

Your mission is to build an army worthy of the Valiant Lion Crusaders. Painted with orange and white stripes. Yeah, yeah, really.

You'll have to implement the elements of your future army, namely a `Squad` and a Tactical Space Marine (`TacticalMarine`).

Let's begin with a `Squad` . Here's the interface you'll have to implement (Include `ISquad.hpp`):

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

Your will implement it so that:

- `getCount()` returns the number of units currently in the squad.
- `getUnit(N)` returns a pointer to the Nth unit (Of course, we start at 0. Null pointer in case of out-of-bounds index.)

- `push(XXX)` adds the XXX unit to the end of the squad. Returns the number of units in the squad after the operation (Adding a null unit, or an unit already in the squad, make no sense at all, of course...)

In the end, the `Squad` we're asking you to create is a simple container of Space Marines, which we'll use to correctly structure your army.

Upon copy construction or assignation of a `Squad`, the copy must be deep. Upon assignation, if there was any unit in the `Squad` before, they must be destroyed before being replaced. You can assume every unit will be created with `new`.

When a `Squad` is destroyed, the units inside are destroyed also, in order.

For `TacticalMarine`, here's the interface to implement (Include `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;
};
```

Constraints:

- `clone()` returns a copy of the current object
- Upon creation, displays: "Tactical Marine ready for battle!"
- `battleCry()` displays: "For the holy PLOT!"
- `rangedAttack()` displays: "* attacks with a bolter *"
- `meleeAttack()` displays: "* attacks with a chainsword *"
- Upon death, displays: "Aaargh..."

Much in the same way, implement an `AssaultTerminator`, with the following outputs:

- Birth: "* teleports from space *"
- `battleCry()` : "This code is unclean. PURIFY IT!"
- `rangedAttack` : "* does nothing *"
- `meleeAttack` : "* attacks with chainfists *"
- Death: "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
$> ./a.out | cat -e
Tactical Marine ready for battle!$
* teleports from space *$
For the holy PLOT!$
* attacks with a bolter *$
* attacks with a chainsword *$
This code is unclean. PURIFY IT!$
* does nothing *$
* attacks with chainfists *$
Aaargh...$
I'll be back...$
```

Be thorough when you're making the `main` function that you will turn in to get your grade...

Chapter V

Exercise 03: Bocal Fantasy

	Exercise : 03
Bocal Fantasy	
Turn-in directory : $ex03/$	
Files to turn in : `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`	
Forbidden functions : `None`	

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/uneqip 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.

Creat 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 it "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* me = new Character("me");

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

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

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

	delete bob;
	delete me;
	delete src;

	return 0;
}
```

Output:

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

Exercise 04: AFK Mining

	Exercise : 04
AFK Mining	
Turn-in directory : *ex*04/	
Files to turn in : `DeepCoreMiner.[hpp,cpp]`, `StripMiner.[hpp,cpp]`, `AsteroKreog.[hpp,cpp]`, `KoalaSteroid.[hpp,cpp]`, `MiningBarge.[hpp,cpp]`, `IAsteroid.hpp`, `IMiningLaser.hpp`, `main.cpp`	
Forbidden functions : `typeid() and more, read the warnings`	

`This exercise does not offer any points but is still useful. You can do it, or not.`

`For this exercise, the use of typeid() is absolutely FORBIDDEN and would result in a -42 for this module. Because that would be cheating, and cheating is 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 kernel 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 beginner, you're gonna need some tools. And since pickaxes are for beginners, 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 `Asteroid` and the `Comet` . 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 `Comet` : "Tartarite"
- `DeepCoreMiner` on `Comet` : "Meium"
- `StripMiner` on `Asteroid` : "Flavium"
- `DeepCoreMiner` on `Asteroid` : "Dragonite"

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.

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.