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 `-42`, 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 d08). 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 `-42`, 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 of the pool, 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.

- The subject can be modified up to 4h before the final turn-in time.

- In case you're wondering, no coding style is enforced during the `C++` pool. 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 day 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

Day-specific rules

- Classes will have to be in Coplien's form, without us asking. It's a pre-requisite to have a positive grade. Be thorough.

Chapter III

Foreword

Here's what Wikipedia has to say on the `Koala`

`The koala (Phascolarctos cinereus or, inaccurately, koala bear) is an arboreal herbivorous marsupial native to Australia. It is the only extant representative of the family Phascolarctidae, and its closest living relatives are the wombats. The koala is found in coastal areas of the mainland's eastern and southern regions, inhabiting Queensland, New South Wales, Victoria and South Australia. It is easily recognisable by its stout, tailless body; round, fluffy ears; and large, spoon-shaped nose. The koala has a body length of 60-85 cm (24-33 in) and weighs 4-15 kg (9-33 lb). Pelage colour ranges from silver grey to chocolate brown. Koalas from the northern populations are typically smaller and lighter in colour than their counterparts further south. It is possible that these populations are separate subspecies, but this is disputed.`

`Koalas typically inhabit open eucalypt woodlands, and the leaves of these trees make up most of their diet. Because this eucalypt diet has limited nutritional and caloric content, koalas are largely sedentary and sleep for up to 20 hours a day. They are asocial animals, and bonding exists only between mothers and dependent offspring. Adult males communicate with loud bellows that intimidate rivals and attract mates. Males mark their presence with secretions from scent glands located on their chests. Being marsupials, koalas give birth to underdeveloped young that crawl into their mothers' pouches, where they stay for the first six to seven months of their life. These young koalas are known as joeys, and are fully weaned at around a year. Koalas have few natural predators and parasites but are threatened by various pathogens, like Chlamydiaceae bacteria and the koala retrovirus, as well as by bushfires and droughts.`

The `Koala` is a cute and funny animal, but sadly, this subject has nothing to do with the `Koala`. Or does it ?

Chapter IV

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`	
Forbidden functions : `None`	
Remarks : `n/a`	

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, who has a name and a title. He has a constructor taking his name and his title as parameters (in this order).

He 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 popped !
```

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 :

```
zaz@blackjack ex00 $ clang++ -W -Wall -Werror *.cpp
zaz@blackjack ex00 $ ./a.out | cat -e
Robert, the Magnificent, is born !$
Some random victim called Jimmy just popped !$
Some random victim called Joe just popped !$
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 !$
zaz@blackjack ex00 $
```

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 V

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 : *ex*01/	
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`	
Forbidden functions : `None`	
Remarks : `n/a`	

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, merge their rectums with their heads, 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 or less, 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* zaz = new Character(``zaz'');

	std::cout << *zaz;

	Enemy* b = new RadScorpion();

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

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

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

	return 0;
}
```

Output :

```
zaz@blackjack ex01 $ clang++ -W -Wall -Werror *.cpp
zaz@blackjack ex01 $ ./a.out | cat -e
zaz has 40 AP and is unarmed$
* click click click *$
zaz has 40 AP and wields a Plasma Rifle$
zaz attacks RadScorpion with a Power Fist$
* pschhh... SBAM! *$
zaz has 32 AP and wields a Power Fist$
zaz has 32 AP and wields a Plasma Rifle$
zaz attacks RadScorpion with a Plasma Rifle$
* piouuu piouuu piouuu *$
zaz has 27 AP and wields a Plasma Rifle$
zaz attacks RadScorpion with a Plasma Rifle$
* piouuu piouuu piouuu *$
* SPROTCH *$
zaz has 22 AP and wields a Plasma Rifle$
```

As usual, turn in a `main` with your tests.