B3 - C++ Pool

B-CPP-300

Day 10

Oh look, a pony!

2.1

Day 10

repository name: *cpp_d10_$ACADEMICYEAR*
repository rights: ramassage-tek
language: C++

- Your repository must contain the totality of your source files, but no useless files (binary, temp files, obj files,...).

All your exercises will be compiled with `g++` **and the** `-Wall -Wextra -Werror` **flags**, unless specified otherwise.

All output goes to the standard output, and must be ended by a newline, unless specified otherwise.

None of your files must contain a `main` function, unless specified otherwise.
We will use our own `main` functions to compile and test your code. It will include your header files.

For each exercise, the files must be turned-in in a separate directory called **exXX** where XX is the exercise number (for instance `ex01`), unless specified otherwise.

Read the examples CAREFULLY. They might require things that weren't mentioned in the subject...

If you do half the exercises because you have comprehension problems, it's okay, it happens. But if you do half the exercices because you're lazy, and leave at 2PM, you **WILL** have problems.
Do not tempt the devil.

The `*alloc`, `free`, `*printf`, `open` and `fopen` functions, as well as the `using namespace` keyword, are forbidden in C++.
By the way, `friend` is forbidden too, as well as any library except the standard one.

UNIT TESTS

It is highly recommended to test your functions as you implement them. It is common practice to create and use what are called **unit tests**.

From now on, we expect you to write unit tests for your functions (when possible). To do so, please follow the instructions in the **"How to write Unit Tests"** document on the intranet, available here.

Exercise 0 - Ponymorphism

Turn in: Sorcerer.hpp, Sorcerer.cpp, Victim.hpp, Victim.cpp, Peon.hpp, Peon.cpp

Polymorphism is an ancient custom, dating back to the time of mages, sorcerers and other charlatans. People will try to make you think they were the first to come up with it, but they're all liars!

Let's take a look at our friend **Ro/b/ert, the Magnificient**, sorcerer by trade.
Robert has an interesting pastime: morphing everything he can get his hands on into sheep, ponies, otters, and many other surprising things.

Let's get started by creating a `Sorcerer` class.
`Sorcerers` have a name and a title.
Their constructor take these `name` and `title` as parameters (in that order).

`Sorcerers` can't be instanciated without parameters (that wouldn't make sense! Imagine a sorcerer with no name or title... poor guy, he couldn't boast to the wenches at the tavern...)

When a `Sorcerer` is born, print:

```
[NAME], [TITLE], is born!
```

Of course, `[NAME]` and `[TITLE]` are to be replaced with the `Sorcerer`'s name and title, respectively...

When a `Sorcerer` dies, print:

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

`Sorcerers` must be able to introduce themselves like so:

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

They can introduce themselves on any **output stream**, thanks to an overload of the << operator.

Remember, the `friend` keyword is forbidden. Add any necessary getters!

Our `Sorcerers` now need victims, to amuse themselves in the morning, between bear claws and troll juice.

Create a `Victim` class.
Much like `Sorcerers`, `Victims` have a name and a constructor taking it as parameter.

When a `Victim` is born, print:

```
Some random victim called [NAME] just popped!
```

When a `Victim` dies, print:

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

A `Victim` can also introduce itself, using the same techniques as `Sorcerers`, and says:

```
I'm [NAME] and i like otters!
```

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

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

While you're at it, add a `void polymorph(const Victim &victim)const` member function to the `Sorcerer` class, so they can now polymorph people.

Now, to add a little variety, our `Sorcerers` would like to polymorph something else, something different from a generic `Victim`. No problem! Let's simply create some more!

Make a `Peon` class.

A `Peon` is a `Victim`.
Which means...?

Upon creation, a `Peon` says:

```
Zog zog.
```

When it dies, it says:

```
Bleuark...
```

Take a look at the sample output, things aren't always as simple as they may seem...

`Peons` get polymorphed like so:

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

It's kind of a poNymorph...

Here is a sample `main` function and its expected output:

```cpp
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;
}
```

Terminal

```
∼/B-CPP-300> g++ -W -Wall -Werror -Wextra -std=c++14 *.cpp
∼/B-CPP-300> ./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!$
```

Exercise 1 - Let them burn

Turn in: AWeapon.hpp/cpp, PlasmaRifle.hpp/cpp, PowerFist.hpp/cpp, AEnemy.hpp/cpp, SuperMutant.hpp/cpp, RadScorpion.hpp/cpp, Character.hpp/cpp

Many things are to be found in the **Wasteland**.
Bits of metal, strange chemicals, crosses, cowboys and homeless wannabe punks…
But mainly, a whole lot of crazy (but funny!) weapons.
It's about time: I'm in the mood to hit stuff today.

To ensure your survival in this God-forsaken place, you're going to have to start coding some weapons.
Complete and implement the following class:

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

private:
    [...]
};
```

A weapon has

- a name,
- a number of damage points dealt when it hits,
- an action point (AP) cost for shooting it.

It produces certain sounds and lighting effects when you `attack` with it.
These side effects are left up to the inheriting classes.

You can now implement the `PlasmaRifle` and `PowerFist` classes.

```
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! *"
```

Now that we have all these shiny new toys, we're going to need some enemies to fight! Or disperse, piledrive, nail to doors, kreogize, merge their rectums with their heads, etc...

Create an `AEnemy` class, based on the following code (which you'll obviously have to complete):

```
class AEnemy
{
public:
    AEnemy(int hp, const std::string &type);
    [...] ~AEnemy();

    virtual void takeDamage(int damage);

    std::string [...] getType() const;
    int getHP() const;

private:
    [...]
};
```

An enemy has a number of hit points and a type.
It can take damage (which reduces its HP). If `damage` is negative, `takeDamage` does nothing.

You can now implement some concrete enemies, so we can have fun with them.

First comes the `SuperMutant`.
Big, bad, ugly and with an IQ usually associated to flowerpots rather than living beings.
That said, it's a bit like a Mancubus in the middle of a hallway: if you miss it, you're doing it on purpose.
That makes it an excellent punching-ball to train with.
Here are its characteristics:

> ***HP***: *170*
> ***Type***: *"Super Mutant"*
> ***Upon birth***, *prints: "Gaaah. Me want smash heads!"*
> ***Upon death***, *prints: "Aaargh..."*

Overloads `takeDamage` to take 3 less damage points than expected (these guys are tough).

While you're at it, create a `RadScorpion`.

> ***HP***: *80*
> ***Type***: *"RadScorpion"*
> ***Upon birth***, *prints: "* click click click *"*
> ***Upon death***, *prints: "* SPROTCH *"*

Now that we have weapons and enemies, all we need is to have a physical appearance of our own!
To do so, create a `Character` class based on the following:

```
class Character
{
public:
    Character(const std::string &name);
    [...]
    ~Character();
    void recoverAP();
    void equip(AWeapon *weapon);
    void attack(AEnemy *enemy);
    std::string [...] getName() const;

private:
    [...]
};
```

A character has

- a name,
- a number of action points (AP),
- a pointer to an `AWeapon` representing its currently equipped weapon.

It starts off with 40 AP and loses AP according to its weapon upon use.
It recovers 10 AP whenever `recoverAP()` is called, up to a maximum of 40.
If it doesn't have enough AP to use a weapon, the attack fails.

It prints

```
[NAME] attacks [ENEMY_TYPE] with a (WEAPON_NAME]
```

when `attack()` is called, and then calls the current weapon's `attack()` method.
If no weapon is equipped, `attack()` does nothing.
The weapon's damage value is then removed from the enemy's HP.
If the target's HP falls to 0 or below, delete it.

`equip()` simply holds a pointer to the weapon.

Overload the << ostream operator to display the attributes of your `Character` (feel free to add any required getters).
The overload must print:

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

If a weapon is equipped.
If not, it prints:

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

Here is a sample `main` function and its expected output:

```cpp
int main()
{
    const auto preda = new Character("Predator");
    const auto prey = new RadScorpion();

    std::cout << *preda;

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

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

    preda->attack(prey);
    std::cout << *preda;
    preda->equip(pr);
    std::cout << *preda;
    preda->attack(prey);
    std::cout << *preda;
    preda->attack(prey);
    std::cout << *preda;

    return 0;
}
```

Terminal

```
~/B-CPP-300> g++ -W -Wall -Werror -Wextra -std=c++14 *.cpp
~/B-CPP-300> ./a.out | cat -e
* click click click *$
Predator has 40 AP and is unarmed$
Predator has 40 AP and wields a Plasma Rifle$
Predator attacks RadScorpion with a Power Fist$
* pschhh... SBAM! *$
Predator has 32 AP and wields a Power Fist$
Predator has 32 AP and wields a Plasma Rifle$
Predator attacks RadScorpion with a Plasma Rifle$
* piouuu piouuu piouuu *$
Predator has 27 AP and wields a Plasma Rifle$
Predator attacks RadScorpion with a Plasma Rifle$
* piouuu piouuu piouuu *$
* SPROTCH *$
Predator has 22 AP and wields a Plasma Rifle$
```

Exercise 2 - PURIFY IT

Turn in: `Squad.hpp/cpp`, `TacticalMarine.hpp/cpp`, `AssaultTerminator.hpp/cpp`

Your mission is to build an army worthy of the **Valiant Lion Crusaders**.
Painted with orange and white stripes. Yeah, really.
You'll have to implement the elements of your future army: a `Squad` and a **Tactical Space Marine** (`TacticalMarine`
).

Let's start with the `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) = 0;
    virtual int push(ISpaceMarine*) = 0;
};
```

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, the first index is 0).
  An out-of-bounds index will result in a null pointer,
- `push(XXX)` adds the `XXX` unit to the end of the squad.
  It returns the number of units in the squad after the operation (adding a null unit or one that is already in the squad makes no sense, 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 assignment to a `Squad`, you must perform **deep copy**.
Upon assignment, if there were units in the `Squad`, they must be destroyed before being replaced.
It is safe for you to assume that every unit will be created with `new`.

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

Concerning `TacticalMarine`, implement the following interface (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;
};
```

`clone()` returns a copy of the current object.

`battleCry()` prints:

```
For the holy PLOT!
```

`rangedAttack()` prints:

```
* attacks with bolter *
```

`meleeAttack()` prints:

```
* attacks with chainsword *
```

Upon creation, it prints:

```
Tactical Marine ready for battle
```

Upon death, it prints:

```
Aaargh...
```

Similarly, implement an `AssaultTerminator`:
`battleCry()` prints:

```
This code is unclean. PURIFY IT!
```

`rangedAttack()` prints:

```
* does nothing *
```

`meleeAttack()` prints:

```
* attacks with chainfists *
```

Upon creation, it prints:

```
* teleports from space *
```

Upon death, it prints:

```
I'll be back...
```

Here is a sample `main` function and its expected output:

```
int main()
{
    std::unique_ptr<ISquad> vlc(new Squad);

    vlc->push(new TacticalMarine);
    vlc->push(new AssaultTerminator);
    for (int i = 0; i < vlc->getCount(); ++i) {
        const auto cur = vlc->getUnit(i);
        cur->battleCry();
        cur->rangedAttack();
        cur->meleeAttack();
    }

    return 0;
}
```

Terminal

```
~/B-CPP-300> g++ -W -Wall -Werror -Wextra -std=c++14 *.cpp
~/B-CPP-300> ./a.out | cat -e
Tactical Marine ready for battle$
* teleports from space *$
For the holy PLOT!$
* attacks with bolter *$
* attacks with chainsword *$
This code is unclean. PURIFY IT!$
* does nothing *$
* attacks with chainfists *$
Aaargh...$
I'll be back...$
```

EXERCISE 3 - KREOG FANTASY VII

Turn in: AMateria.hpp/cpp, Ice.hpp/cpp, Cure.hpp/cpp, Character.hpp/cpp, MateriaSource.hpp/cpp

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

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

    const std::string &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);

private:
    [...]
    unsigned int xp_;
};
```

A **Materia** has a total XP which starts at 0, and increases by 10 when the Materia is used.

Think of a smart way to do so!

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

Their clone() method returns, of course, a new instance of the real Materia's type.
The use(ICharacter &target) method must display the following, respectively for Ice and Cure:

```
* shoots an ice bolt at [NAME] *
* heals [NAME]'s wounds *
```

Of course, you must replace [NAME] with the name of target.

When assigning one Materia to another, copying the type doesn't make sense...

Create the `Character` class which must implement the following interface (include `ICharacter.hpp`):

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

A `Character` has an inventory of 4 Materia at most, which starts off empty.
A `Character` will equip Materia in its inventory slots 0 to 3, in order.
If a `Character` tries to equip a Materia with a full inventory, or uses/unequips a nonexistent Materia, you shouldn't do anything.

The `unequip` method must NOT delete Materia!

The `use` method uses the Materia at the `idx` slot, targeting `target`.

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 assignment to a `Character` must be deep, of course.
The old `Materia` of a `Character` must be deleted.
The same applies when destroying a `Character`.

Now that your `Characters` can equip and use `Materia`, things are starting to look right.

That said, I would hate to have to create `Materia` by hand, and always have to know their real type...
To avoid this problem, create a smart **Source of Materia**.

Create the `MateriaSource` class, which must implement the following interface (include `IMateriaSource.hpp`):

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

`learnMateria` must copy the `Materia` passed as parameter and hold it in memory, so that it can be clloned later.
Much in the same way as `Characters`, the `Source` can know at most 4 `Materia` at any given time.
`createMateria` returns a new `Materia`, which is be a copy of the previously learned `Materia` with a `type` matching the parameter.

The function returns a null pointer if `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 `Materias` without knowing their “real” type, just a string identifying them.

This is an implementation of the **Abstract Factory** pattern.
Look it up!

Here is a sample `main` function and its expected output:

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

    std::unique_ptr<ICharacter> perceval(new Character("Perceval"));

    auto tmp = src->createMateria("ice");
    perceval->equip(tmp);
    tmp = src->createMateria("cure");
    perceval->equip(tmp);

    std::unique_ptr<ICharacter> bohort(new Character("Bohort"));

    perceval->use(0, *bohort);
    perceval->use(1, *bohort);

    return 0;
}
```

Terminal

```
~/B-CPP-300> g++ -W -Wall -Werror -Wextra -std=c++14 *.cpp
~/B-CPP-300> ./a.out | cat -e
* shoots an ice bolt at Bohort *$
* heals Bohort's wounds *$
```

Exercise 4 - KreogSwarm

Turn in: DeepCoreMiner.hpp/cpp, StripMiner.hpp/cpp, AsteroKreog.hpp/cpp, KoalaSteroid.hpp/cpp,

MiningBarge.hpp/cpp, IAsteroid.hpp
Forbidden features: typeid()

At first glance, you might think that the space beyond the **KoalaGate** is just vast nothingness.
But no, good sir.
It's actually home to a metric fuckton of random useless stuff.

Between Space Bimbos, hideous monsters, space trash and even a few Microsoft developers, you'll find an incredible amount of asteroids there, each filled with minerals more precious than the last.
A little bit like the goldrush, but without Scrooge McDuck.

And 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 (include IMiningLaser.hpp):

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

Implement the DeepCoreMiner and StripMiner concrete lasers.
Their mine method must produce the following output, respectively for DeepCoreMiner and StripMiner

```
* mining deep... got [RESULT]! *
* strip mining... got [RESULT]! *
```

You must replace [RESULT] with the return value of the asteroid's beMined function.

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

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

The two asteroids to implement are the AsteroKreog and the KoalaSteroid.
Their getName method must return their name (you don't say?), which is the same as the class name.

Using inheritance and parametric polymorphism (and your brain, hopefully), make it so that a call to `IMiningLaser::mine` yields a result depending on the type of asteroid AND the type of laser.

The return values must be as follows:

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

To do so, you'll have to complete the `IAsteroid` interface.

You probably need two `beMined` methods...
They would take their parameter by non-const pointer, and would both be const.
Don't add anything else.

Don't try to deduce the return value based on the asteroid's `getName` function.
You NEED to use TYPES and POLYMORPHISM.
Any other devious way (`typeid`, `dynamic_cast`, `getName` etc...) is strictly forbidden.

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 *laser);
    void mine(IAsteroid *asteroid) const;
};
```

A barge starts with no laser and can equip up to 4 of them.
If it already has 4 lasers, `equip` does nothing.

We don't copy.

The `mine` method calls `IMiningLaser::mine` on all the equipped lasers, in the order they were equipped in.

You don't need a main function by now, do you?
You're big boys now.