B3 - C++ Pool

B-CPP-300

Day 07 afternoon

SKAT

1.25

{EPITECH.}

Day 07 afternoon

binary name:	no binary
group size:	1
repository name:	cpp_d07a
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** `-W -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.

Create a directory named `tests`. For each of the functions you turn in, create a file in that directory named `tests-FUNCTION_NAME.c` containing all the tests needed to cover all of the exercise's possible cases (regular or irregular).

Here is a sample set of unit tests for the **string** class:

```
#include <criterion/criterion.h>

Test(string, default_value)
{
        std::string s;
        cr_assert_eq(s, "");
}

Test(string, assign)
{
        std::string s;

        s = "test";
        cr_assert_eq(s, "test");
}

Test(string, append)
{
        std::string s("test");

        s += "ing";
        cr_assert_eq(s, "testing");
}
```

EXERCISE 0 - MEEEEEEEDIC

Turn in: `Skat.hpp`, `Skat.cpp`
Forbidden features: pointers

Soldiers, welcome to the **SKAT (Special Kreog And Tactical)**.
You are here because you want to protect our wonderful country, Australia, from the crawling brood that are **Dingos**.
This infamous rot won't rest until they undermine the core foundation of our wonderful country.

I am **drill sergeant Hartog** and I am in charge of making war dogs out of you.

However, you are now at level zero of your life on earth.
It's time for you to rack in some experience before you can **show them what you've got**.

Let's start the briefing.
As you will very likely get a bullet in your skin on the battlefield, our team of scientists have designed a special package that will save your butt more than you would think: the **stimpak**.
This stimpak can repair your bones, suture clean wounds, and so on.
As long as you have one left, you have a chance to stay alive and come back home.

Implement the following class:

```
class Skat
{
public:
        Skat(const std::string &name, int stimPaks);
        ~Skat();

        [...] stimPaks();
        const std::string &name();

        void shareStimPaks(int number, [...] stock);
        void addStimPaks(unsigned int number);
        void useStimPaks();
        void status();

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

A Skat has a name, represented by a `string`, and a number of stimpaks.
By default, your Skat's name is **bob**, and it has 15 stimpaks.

The `stimPaks` member function returns the number of stimpaks your Skat currently has.
It is possible to modify the number of stimpaks your unit has by calling this function.

The `name` member function returns your unit's name.
It doesn't modify the calling instance.

The `shareStimPaks` member function lets you provide extra stimpaks to a teammate in need.
It increments by `number` the `stock` of stimpaks.

After doing so, it decrements your stock by `number` stimpaks.
If the number of stimpaks required is too big, print:

```
Don't be greedy
```

on the standard output, and do nothing.
Otherwise, print:

```
Keep the change.
```

The `addStimPaks(unsigned int number)` member function adds `number` stimpaks to your unit's collection.
If `number` is equal to 0, it prints:

```
Hey boya, did you forget something?
```

Your unit can use stimpaks by calling the `useStimPaks()` member function.
It prints:

```
Time to kick some ass and chew bubble gum.
```

when possible.
Otherwise, it prints:

```
Mediiiiiic
```

You can query a unit's status at any point by calling its `status` member function.
A unit communicates like so:

```
Soldier [NAME] reporting [NUMBER] stimpaks remaining sir!
```

where `[NAME]` is the name of the unit and `[NUMBER]` its number of stimpaks.
This member function can be called on immutable `Skats`.

The following code must compile and display the following output:

```cpp
int main()
{
        Skat s("Junior", 5);

        std::cout << "Soldier " << s.name() << std::endl;
        s.status();
        s.useStimPaks();

        return 0;
}
```

```
Terminal
~/B-CPP-300> g++ -W -Wall -Werror -Wextra -std=c++14 *.cpp
~/B-CPP-300> ./a.out | cat -e
Soldier Junior$
Soldier Junior reporting 5 stimpaks remaining sir!$
Time to kick some ass and chew bubble gum.$
```

EXERCISE 1 - KOALABOT

Turn in: `KoalaBot.hpp`, `KoalaBot.cpp`, `Parts.hpp`, `Parts.cpp`

Gentlemen, this is the **KoalaBot Kreog mk5** (the 4 previous versions have a tendency to burst into flames every now and then).
This new prototype will be your backup plan when the situation gets too tough for you (a recon mission in a mine field, for example).

The KoalaBot has 3 removable parts: arms, legs, and a head.
I'll break the knees of the first of you to tell me the bot has 2 arms and legs.

All these parts are built according to the following model:

- Each class has a constructor with the following signature:

```
Constructor(std::string [...] serial, bool functional);
```

- They must have a serial represented by a `string`, as well as a boolean indicating whether or not it is functional:

  - The `Arms` class has a default serial of *"A-01"*.
  - The `Legs` class has a default serial of *"L-01"*.
  - The `Head` class has a default serial of *"H-01"*.

  These classes are all functional by default.

- They should have the following public member functions:

  - `bool isFunctionnal()` indicates whether the piece is functional or not,
  - `std::string serial()` returns the part's serial,
  - `void informations()` prints:

```
[Parts] [PARTSTYPE] [SERIAL] status : {OK | KO}
```

    preceded by a single tab, and followed by a newline.

    `[PARTSTYPE]` must be replaced by the name of the class.
    `[SERIAL]` must be replaced by the object's serial.
    If the part is functional, print *"OK"*. Otherwise, print *"KO"*.

None of these member functions modify the calling instance.

Now that we have the various pieces, let's put them together.
Create the `KoalaBot` class.

- it is composed of an instance of each of the previously created parts (`Arms`, `Legs` and `Head`),
- it has a serial, represented by a `string`, with a default value of *"Bob-01"*,
- it has a `setParts` member function that takes a reference to a constant part as parameter;

This function can be called with any class from the `Parts.hpp` file.

- it has a `swapParts` member function that takes a reference to a part as parameter.
  This part will be swapped with the `KoalaBot`'s.

This function can be called with any class from the `Parts.hpp` file.

- it has a `void informations()` member function that prints:

```
[KoalaBot] [SERIAL]
```

  followed by a newline,
  `[SERIAL]` is the `KoalaBot`'s serial, followed by information about each part of the `KoalaBot`,
  in the following order `Arms`, `Legs` then `Head`.

This member function does not modify the calling instance.

- it has a `bool status()` member function that returns `true` if all the parts of the `KoalaBot` are running, and `false` otherwise.

This member function does not modify the calling instance.

The following code must compile and print the following output:

```
int main()
{
        KoalaBot kb;

        std::cout << std::boolalpha << kb.status() << std::endl;
        kb.informations();

        return 0;
}
```

Terminal

```
~/B-CPP-300> g++ -W -Wall -Werror -Wextra -std=c++14 *.cpp
~/B-CPP-300> ./a.out | cat -e
true$
[KoalaBot] Bob-01$
     [Parts] Arms A-01 status :  OK$
     [Parts] Legs L-01 status :  OK$
     [Parts] Head H-01 status :  OK$
```

Exercise 2 - Houston, we have a problem

Turn in: `KreogCom.hpp`, `KreogCom.cpp`

You are a team, and as such are closer than Parisians in the subway during rush hour.
This is true whether you are enjoying well-deserved R&R in the barracks, or deployed all over the jungle.
In order to communicate, and more importantly locate your teammates, I introduce to you the pinnacle of our technology: the `KreogCom`.
This technological jewel locates your teammates in real-time.

The `KreogCom` works in the following way:

```
class KreogCom
{
public:
        KreogCom(int x, int y, int serial);
        ~KreogCom();

        void addCom(int x, int y, int serial);
        void removeCom();
        KreogCom *getCom();

        void ping();
        void locateSquad();

private:
        const int m_serial;
};
```

A `KreogCom` has a serial represented by a constant integer, as well as `x` and `y` coordinates.

It is up to you to add member data, but it must be private.

It has a constructor that lets you create a `KreogCom` by specifying its `x` and `y` coordinates, as well as its `serial`.

It has an `addCom` member function that creates a new `KreogCom`.
If the current `KreogCom` is not linked to any `KreogCom`, it is linked to the newly created one:

```
+------+     +---------------------------+
| this | --> | new KreogCom x, y, serial |
+------+     +---------------------------+
```

If the current `KreogCom` is already linked to another `KreogCom`, then the new `KreogCom` will replace it (see below):

```
+------+     +---------------------------+     +----------------------------------+
| this | --> | new KreogCom x, y, serial | --> | KreogCom that was linked to this |
+------+     + --------------------------+     +----------------------------------+
```

It has a `getCom` member function that returns a pointer to the `KreogCom` linked to the current instance. If it is not linked, returns a null pointer.

It has a `removeCom` member function that removes the linked `KreogCom`.

Be careful not to break the communication chain.

It has a `ping` member function that prints the following information to the standard output:

```
KreogCom [SERIAL] currently at [X] [Y]
```

`[SERIAL]`, `[X]` and `[Y]` are the member data of your `KreogCom`.

This member function doesn't modify the calling instance.

It has a `locateSquad` member function that prints information about all the linked `KreogCom`, and then the current instance's information:

```
[Displays info on linked KreogCom]
[Displays info on current KreogCom]
```

This member function doesn't modify the calling instance.

When destroying a `KreogCom`, the communication chain must not be broken.

The following code must compile and print the following output:

```
int main()
{
        KreogCom k(42, 42, 101010);

        k.addCom(56, 25, 65);
        k.addCom(73, 34, 51);

        k.locateSquad();

        k.removeCom();
        k.removeCom();

        return 0;
}
```

Terminal

```
~/B-CPP-300> g++ -W -Wall -Werror -Wextra -std=c++14 *.cpp
~/B-CPP-300> ./a.out | cat -e
KreogCom 101010 initialized$
KreogCom 65 initialized$
KreogCom 51 initialized$
KreogCom 51 currently at 73 34$
KreogCom 65 currently at 56 25$
KreogCom 101010 currently at 42 42$
KreogCom 51 shutting down$
KreogCom 65 shutting down$
KreogCom 101010 shutting down$
```

EXERCISE 3 - LOCK'N LOAD, BABY

Turn in: `Phaser.hpp`, `Phaser.cpp`, `Sounds.hpp`

Now that you've gone through basic training, it's time to have some real fun.
This is the **Phaser Kreog'o Blaster mk2**, your new best friend.
From now on, you will train with it, eat with it and sleep with it.
This sweetheart has three different firing modes:

- `REGULAR`, for clean jobs
- `PLASMA`, to warm up the mood
- `ROCKET`, for surgical strikes

However, due to budget cuts, the weapon is being delivered in spare parts.
It's time to roll up your sleeves and get to work, if you ever want to get blasting Dingos.

Implement the following class:

```
class Phaser
{
public:
        enum AmmoType
        { ... };

        Phaser(int maxAmmo, AmmoType type);
        ~Phaser();

        void fire();
        void ejectClip();
        void changeType(AmmoType newType);
        void reload();
        void addAmmo(AmmoType type);

        int getCurrentAmmos();

    private:
        static const int Empty;
        [...]
};
```

The `Sounds` class must define the following constant class variables:

```
std::string Regular;
std::string Plasma;
std::string Rocket;
```

These variables must not be assigned within the files you turn in!
That will be done in the correction `main`.

A `Phaser` has a maximum amount of ammunition, a number representing the amount of ammunition currently loaded, a sound for each type of firing mode, a magazine with ammunition and a default type of ammunition for the weapon.

A `Phaser` has a default magazine of 20 `REGULAR` bullets.

A `Phaser` is fully loaded upon creation.

The `Empty` variable represents an empty magazine.
It is strongly recommended to use it in your program.
You MUST initialize it with the value 0.

When calling the `fire` member function, you must print

```
Clip empty, please reload
```

if the magazine is empty.
If not, you must print the sound of the ammunition loaded in the first case of the magazine.
Once done, the size of the magazine is reduced by 1, to represent the round that was fired.

The `ejectClip` member function ejects the magazine in the weapon and reduces the amount of ammunition to 0.

The `changeType` member function prints the following to the standard output:

```
Switching ammo to type: [TYPE]
```

where `[TYPE]` is the value of the parameter in lowercase (ex: `regular` for `REGULAR`).
Calling this member function changes the default type for the `Phaser`.

The `reload` member function prints

```
Reloading...
```

and reloads the weapon with its default ammunition type.

The `addAmmo` member function adds a round with the `type` type at the end of the magazine.
If the current amount of ammunition is equal to the maximum amount for the weapon, print

```
Clip full
```

to the standard output, and do nothing.

The `getCurrentAmmos` member function returns the amount of ammunition in the magazine.

The `getCurrentAmmos` member function can be called on immutable `Phaser` objects.

The following code must compile and print the following output:

```
int main()
{
        Phaser p(5, Phaser::ROCKET);

        p.fire();
        p.reload();

        std::cout << "Firing all ammunition" << std::endl;
        while (p.getCurrentAmmos() > 0)
                p.fire();

        return 0;
}
```

Terminal

```
~/B-CPP-300> g++ -W -Wall -Werror -Wextra -std=c++14 *.cpp
~/B-CPP-300> ./a.out | cat -e
Boooooooom$
Reloading...$
Firing all ammunitions$
Boooooooom$
Boooooooom$
Boooooooom$
Boooooooom$
Boooooooom$
```

In this example, the rocket sound is “Boooooooom”. It’s up to you to find out how to initialize it.

EXERCISE 4 - G-SQUAD

Turn in: `Skat.hpp/cpp`, `Phaser.hpp/cpp`, `Sounds.hpp`, `KreogCom.hpp/cpp`, `Squad.hpp/cpp`

OK guys, you're all set.
I have nothing more to teach you.
However, before you go blasting Dingos, you have to get geared up.

A `Skat` has a `Phaser` and a `KreogCom`.
Thus, add a constructor with the following signature:

```
Skat(const std::string &name, int stimPaks, int serial, int x, int y,
                                                    Phaser::AmmoType type);
```

The `serial`, `x`, `y` and `type` parameters are the information required to construct the`KreogCom` and `Phaser`.

Each `Skat` gets a `Phaser` with 20 ammunition.

This class doesn't have a default constructor anymore.

Add the following member functions:

- `void fire()` so that the `Skat` opens fire
- `void locate()` gives the `Skat`'s position
- `void reload()` so that the `Skat` reloads
- `KreogCom &com()` provides access to the `Skat`'s communicator

It's up to you to figure out which of these member functions are constant.

Add the following member function to the `KreogCom`: `void addCom(KreogCom *com)`, which links the current `KreogCom` to `com`.

The correction won't call the `removeCom` member function, no need to modify it.

Good.
Now that you look like something decent, it's time to get organized.

Implement the following class:

```
class Squad
{
public:
        Squad(int posXBegin, int posYBegin, Phaser::AmmoType ammoType,
              int size);
        ~Squad();

        void fire();
        void localisation();

        [...] skats();
        [...] at(int idx);
        int size();

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

Sounds from the `Sounds` class must be initialized in the `Squad.cpp` file, with the following values:

- Regular: *"Bang"*
- Plasma: *"Fwooosh"*
- Rocket: *"Boouuuuuum"* !!!

A squad is composed of `size` `Skats`. By default, a squad has 5 `Skats`.

The last teammate in a squad must be a null pointer.

Teammate's communicators serials are equivalent to the place of the `Skat` within the squad (`Skat` 0 -> com `serial` 0, ..., `Skat` n -> com `serial` n).

The first `Skat` is located at the `posXBegin`, `posYBegin` position.

The `x` position is incremented by 10 for each `Skat` of the squad.
The `y` position is incremented by 15 for each `Skat` of the squad.
For instance,

```
Skat 0 -> X = 0, Y = 0;
Skat 1 -> X = 10, Y = 15;
...;
Skat n -> X = (n) * 10, Y = (n) * 15
```

The communicator of the `Skats` are linked together, based on the following diagram:

```
+-------------+     +-------------+     +-----+     +-------------+
| com Skat 0 | --> | com Skat 1 | --> | ... | --> | com Skat n |
+-------------+     +-------------+     +-----+     +-------------+
```

The `fire` member function makes every squad member open fire.

The `localisation` member function prints the position of each squad member, starting from the FIRST member

The `skats` member function returns all the squad members

The `at` member function returns the `Skat` at the `idx` index. If `idx` is not valid, the function returns a null pointer

The `size` member function returns the size of the squad

In order to simplify squad interaction, **implement the following function**:

```
void foreach([...] beginIdx, [...] actionPtr)
```

in the files related to the squad.
This function takes as parameter an index representing the beginning of a group of `Skats`, as well as a pointer to a member function of the `Skat` class, taking no parameters and returning `void`.

This function will be called with all the member functions of the `Skat` class with no parameters and returning `void`.

Is a member function different from a const member function with the exact same signature? int class::fct() ==? int class::fct() const

The following code must compile and print the following:

```
int main()
{
        Squad s(0, 0, Phaser::REGULAR);

        s.fire();

        return 0;
}
```

Terminal

```
~/B-CPP-300> g++ -W -Wall -Werror -Wextra -std=c++14 *.cpp
~/B-CPP-300> ./a.out | cat -e
KreogCom 0 initialized$
KreogCom 1 initialized$
KreogCom 2 initialized$
KreogCom 3 initialized$
KreogCom 4 initialized$
Bang$
Bang$
Bang$
Bang$
Bang$
KreogCom 0 shutting down$
KreogCom 1 shutting down$
KreogCom 2 shutting down$
KreogCom 3 shutting down$
KreogCom 4 shutting down$
```

That's it.
Break now private, and good luck for your mission.