B3 - C++ Pool

B-CPP-300

Day 13

A Game of Toys

1.15

Day 13

binary name:	no binary
group size:	1
repository name:	cpp_d13
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 - Encapsulation

Turn in: `Picture.hpp`, `Picture.cpp`, `Toy.hpp`, `Toy.cpp`

You are going to create some basic toys for you to play with, each with a picture (so you can know what it looks like!).
More features will be added to these toys in the following exercises.

Start by creating a `Picture` class to represent our toys' illustrations.
It must contain, publicly:

```
1. std::string data;
```

Our toy's ASCII art.

```
2. bool getPictureFromFile(const std::string &file);
```

Sets `data`'s value to the content of `file`.
If an error occurs, `data` must be set to *"ERROR"* and the function must return `false`.
Otherwise, it returns `true`.

```
3. Picture(const std::string &file);
```

Creates a `Picture` object by loading the content of `file`.
If an error occurs, `data` must be set to *"ERROR"*.
Creating a `Picture` without a filename as parameter sets `data` to an empty string.

Now, create a `Toy` class.
It must contain a `ToyType` enumeration with two fields: `BASIC_TOY` and `ALIEN`.

The `Toy` class must contain a type, a name and a picture, as well as the following member functions:

- `getType`, a getter for the toy's type (there is no setter, as the type will never change),
- `getName`,
- `setName`,
- `setAscii` that takes a filename as parameter and sets the toy's picture to the file's content. Returns `true` if it succeeds, `false` otherwise,
- `getAscii` that returns the toy's picture as a `string`,
- a constructor taking no parameter, setting the toy's type to `BASIC_TOY`, its name to *"toy"* and its picture to an empty `string`,
- a constructor taking three parameters: the `ToyType`, a `string` containing the toy's name, and a `string` containing the picture's filename.

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

```
#include <iostream>
#include "Toy.hpp"

int main()
{
        Toy toto;
        Toy ET(Toy::ALIEN, "green", "./alien.txt");

        toto.setName("TOTO !");

        if (toto.getType() == Toy::BASIC_TOY)
                std::cout << "basic toy: " << toto.getName() << std::endl
                        << toto.getAscii() << std::endl;
        if (ET.getType() == Toy::ALIEN)
                std::cout << "this alien is: " << ET.getName() << std::endl
                        << ET.getAscii() << std::endl;
        return 0;
}
```

Terminal

```
~/B-CPP-300> ./a.out
basic toy:  TOTO !

this alien is:  green

          _|_
    ,_.-_,  _  ,_-._,
     \ (.)(.)(.) /
  _,   `\_-===-_/`   ,_
>   |----"""""----|   <
`""`--/    _-@-\--`""`
      |===L_I===|
       \       /
      _\__|__/_
      `"""""`"""`
```

Exercise 1 - Canonical form

Turn in: `Picture.hpp`, `Picture.cpp`, `Toy.hpp`, `Toy.cpp`

Re-use the two classes from the previous exercise and make them comply with the canonical form.

This may imply more than meets the eye…

Exercise 2 - Simple inheritance

Turn in: `Picture.hpp/cpp`, `Toy.hpp/cpp`, `Buzz.hpp/.cpp`, `Woody.hpp/cpp`

Add two values to the `ToyType` enumeration: `BUZZ` and `WOODY`, and create two new `Buzz` and `Woody` classes.

These two classes inherit from `Toy`, and set their parent's attributes to the corresponding values upon construction:

- `type`: `BUZZ` and `WOODY`, respectively
- `name`: passed as parameter
- `ascii`: optionally passed as parameter.
  If no filename is provided, the objects will respectively load their picture from the *"buzz.txt"* and *"woody.txt"* files.

It shouldn't be possible to create `Buzz` or `Woody` objects without a name.

EXERCISE 3 - PONYMORPHISM

Turn in: `Picture.hpp/cpp`, `Toy.hpp/cpp`, `Buzz.hpp/.cpp`, `Woody.hpp/cpp`

We'd like our toys to be able to speak.
Add a `speak` method to the `Toy` class, taking the statement to say as a parameter.

This method displays the toy's name, followed by a space and the statement passed as parameter.

```
[NAME] "[STATEMENT]"
```

Overload this method in the `Buzz` and `Woody` classes in order to display (respectively):

```
BUZZ: [NAME] "[STATEMENT]"
WOODY: [NAME] "[STATEMENT]"
```

In all three cases, [NAME] is to be replaced with the toy's name and [STATEMENT] with the `string` passed as parameter.

The double quotes in the examples must be printed.

The `speak` method must **not** be `const`.
You'll understand why in the following exercises.

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

```
#include <iostream>
#include "Toy.hpp"
#include "Buzz.hpp"
#include "Woody.hpp"

int main()
{
        std::unique_ptr<Toy> b(new Buzz("buzziiiii"));
        std::unique_ptr<Toy> w(new Woody("wood"));
        std::unique_ptr<Toy> t(new Toy(Toy::ALIEN, "ET", "alien.txt"));

        b->speak("To the code, and beyond !!!!!!!!");
        w->speak("There's a snake in my boot.");
        t->speak("the claaaaaaw");
}
```

Terminal

```
~/B-CPP-300> ./a.out
BUZZ: buzziiiii "To the code, and beyond !!!!!!!!"
WOODY: wood "There's a snake in my boot."
ET "the claaaaaaw"
```

EXERCISE 4 - OPERATORS

Turn in: `Picture.hpp/cpp`, `Toy.hpp/cpp`, `Buzz.hpp/.cpp`, `Woody.hpp/cpp`

You will now add two operator overloads.

A first overload of the `<<` operator, between an `ostream` and a `Toy`.
This operator will print the toy's name, followed by its picture, on the standard output.
The name and picture will have to be followed by a newline.

A second overload the `<<` operator, between a `Toy` and a `string`.
This operator will replace the `Toy`'s picture with the `string`.

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

```cpp
#include <iostream>
#include "Toy.hpp"

int main()
{
        Toy a(Toy::BASIC_TOY, "REX", "rex.txt");

        std::cout << a;
        a << "\o/";
        std::cout << a;
}
```

Terminal

```
~/B-CPP-300> ./a.out
REX
                              _.-.
                             /  99\
                            (      '\
                            |\ ,   ,|
                    __      | \____/
              ,.--"'-."."   /   '---'
          _.-'          '-/      |
      _.-"   |   '-.             |_/_
,__.-'  _,.--\      \      ((    /-\
',_..--'      `\     \      \_ /
                `-,   )      |\'
                  |   |-.,,-" (
                  |   |   '\   '',_
                  )    \    \,(\(\-'
                  \     '-,_
                   \_(\-(\'-'
                      "  "

REX
\o/
```

Exercise 5 - Nesting

Turn in: `Picture.hpp/cpp`, `Toy.hpp/cpp`, `Buzz.hpp/.cpp`, `Woody.hpp/cpp`

We know some toys have several options: for example, our Buzz Lightyear toy can speak spanish!
To illustrate this, add a `speak_es` method to the `Toy` class, with the same signature as `speak`.
In the `Buzz` class, this method must have the same behavior as `speak` but must add *"senorita"* before and after the statement:

```
BUZZ: name senorita "statement" senorita
```

However, some toys don't speak spanish, so we have to handle this case.
For every toy that can't speak spanish, the `speak_es` method doesn't display anything and returns `false`.

Let's make the most of our error handling in the `Toy` class.
We currently have two possible error causes:

- `setAscii`
- `speak_es`

Both return `false` in the event an error occured.

Create a nested `Error` class in `Toy` that contains two methods and a public attribute:

- `what`: returns the error message:
  - *"bad new illustration"* if the error happened in `setAscii`
  - *"wrong mode"* if the error happend in `speak_es`
- `where`: returns the name of the function where the error occured,
- `type`: holds the error type.

Moreover, `Error` must contain an `ErrorType` enum with the different error types:

- `UNKNOWN`
- `PICTURE`
- `SPEAK`

Add a `getLastError` to the `Toy` class that will return an `Error` object containing information about the last error that occured.
If no error happened, `getLastError` returns an `Error` instance with two empty strings for `what` and `where`, and has `UNKNOWN` as its type.

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

```
#include <iostream>
#include "Toy.hpp"
#include "Buzz.hpp"
#include "Woody.hpp"

int main()
{
        Woody w("wood");

        if (w.setAscii("file_who_does_not_exist.txt") == false)
        {
                auto e = w.getLastError();
                if (e.type == Toy::Error::PICTURE)
                {
                        std::cout << "Error in " << e.where() << ": " << e.what() <<
                            std::endl;
                }
        }

        if (w.speak_es("Woody does not have spanish mode") == false)
        {
                auto e = w.getLastError();
                if (e.type == Toy::Error::SPEAK)
                {
                        std::cout << "Error in " << e.where() << ": " << e.what() <<
                            std::endl;
                }
        }

        if (w.speak_es("Woody does not have spanish mode") == false)
        {
                auto e = w.getLastError();
                if (e.type == Toy::Error::SPEAK)
                {
                        std::cout << "Error in " << e.where() << ": " << e.what() <<
                            std::endl;
                }
        }
}
```

Terminal

```
~/B-CPP-300> ./a.out
Error in setAscii:  bad new illustration
Error in speak_es:  wrong mode
Error in speak_es:  wrong mode
```

EXERCISE 6 - A TOY STORY

Turn in: `Picture.hpp/cpp`, `Toy.hpp/cpp`, `Buzz.hpp/.cpp`, `Woody.hpp/cpp`, `ToyStory.hpp/cpp`

Create a `ToyStory` class which tells stories about two toys.

`ToyStory` contains a class `tellMeAStory` that takes 5 parameters:

- a filename containing the story,
- the first `Toy`, which we'll call `toy1`,
- a `Toy` method pointer taking a `string` as parameter and returning a boolean, which we'll call `func1`,
- the second `Toy`, which we'll call `toy2`,
- a `Toy` method pointer taking a `string` as parameter and returning a boolean, which we'll call `func2`.

These `Toy` instances and method pointers are respectively associated.
`toy1` is associated with `func1` and `toy2` is associated with `func2`.

The `tellMeAStory` function starts by printing the two `Toys`' pictures, each followed by a newline.
It then reads the file given as parameter, and for each line in it, calls the method pointer associated to the toy.
The toys will be called on a rotating basis:

- the first line will be sent to `func1` on `toy1`,
- the second to `func2` on `toy2`,
- the third to `func1` on `toy1`,
- ...

If the line starts with *"picture:"*, it changes the picture of the toy which was supposed to be called.
The toy's new picture is then set to the content of the file specified after the *"picture:"* mention.
The toy's picture is then displayed.

For instance, with the following file:

Terminal

```
~/B-CPP-300> cat story.txt
hi
picture:ham.txt
nothing special
CU
```

The actions must be the following:

- print `toy1`'s picture followed by a newline,
- print `toy2`'s picture followed by a newline,
- call `func1` on `toy1` with *"hi"*,
- set `toy2`'s picture to the content of the *"ham.txt"* file,
- print `toy2`'s picture,
- call `func2` on `toy2` with *"nothing special"*,
- call `func1` on `toy1` with *"CU"*.

`tellMeAStory` stops as soon as it encounters an error (if it fails to change a toy's picture, for instance).
If an error occurs, print information about it using the following format:

```
where: what
```

`where` must be replaced with the error's `where` property, and `what` must be replaced with the error's `what` property.

If the file passed as parameter cannot be opened or read, print *"Bad Story"* to the standard output.

Here is a sample `main` function:

```
#include <iostream>
#include "Toy.hpp"
#include "ToyStory.hpp"
#include "Buzz.hpp"
#include "Woody.hpp"

int main()
{
        Buzz  b("buzzi");
        Woody w("wood");

        ToyStory::tellMeAStory("superStory.txt", b, &Toy::speak_es, w, &Toy::speak);
}
```