B3 - C++ Pool

B-PAV-242

Day 14 - Afternoon

Everything is clear, except...

KOALA

42.0

Day 14 - Afternoon

binary name:	no binary
group size:	1
repository name:	cpp_d14a
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,...).

General Setpoints

READ THESE CAREFULLY
You will have no possible excuse if you end up with a 0 because you didn't follow one of these.

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.

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

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

Remember: you're coding in C++ now, and not in C. Therefore, the following functions are **FORBIDDEN** and their use will be punished by a -42, no questions asked:

*alloc
*printf
free

Any use of the `friend` keyword will result in a -42

You are not allowed to use any library other than the C++ standard library.

It must be possible to include each of your header files independently from the others. Headers must include all their dependencies.

All your header files will be included in the correction `main`.

None of your files must contain a `main` function

THINK. Please.

THINK

T.H.I.N.K.! For Pony!

To avoid compilation problems during automated tests, please include all necessary files within your headers.

Please note that 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.

This subject may be modified up to one hour before turn-in time!

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 classes you turn in, create a file in that directory named `tests-CLASS-NAME.cpp` containing all the tests needed to cover all of the class' 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 - Errors

KOALA	Exercise: 00	points : 4
Errors		
Turn-in directory: cpp_d14a/ex00		
Compiler: g++	Compilation flags: -W -Wall -Wextra -Werror -std=c++14	
Makefile: No	Rules: n/a	
Files to turn in: Errors.hpp, Errors.cpp		
Notes: The ‘Errors.hpp‘ file is provided		
Forbidden functions: ‘using namespace‘ keyword		

Welcome to **NASA**! No time to explain I'm afraid, but as of now you are working on the **Mars Rover** prototype. Your first mission is to implement the error reporting system. Errors will have to comply with the following inheritance tree:

- `std::exception`
  - `NasaError`
    - `LifeCriticalError`
    - `MissionCriticalError`
    - `CommunicationError`
    - `UserError`

The exceptions' `getComponent()` method should return the name of the component, which they receive as their second constructor parameter. Note that `CommunicationError`'s `getComponent` method should always return *"CommunicationDevice"*.

`getComponent()` must have the following prototype:

```
const std::string &getComponent() const;
```

Exercise 1 - Tests

	Exercise: 01	points : 4
Tests		
Turn-in directory: `cpp_d14a/ex01`		
Compiler: `g++`	Compilation flags: `-W -Wall -Wextra -Werror -std=c++14`	
Makefile: `No`	Rules: `n/a`	
Files to turn in: `Makefile Errors.hpp, Errors.cpp, BaseComponent.hpp, Engine.cpp, Engine.hpp, Oxygenator.hpp, Oxygenator.cpp, AtmosphereRegulator.hpp, AtmosphereRegulator.cpp, WaterReclaimer.hpp, WaterReclaimer.cpp`		
Notes: `Your objective is to have ‘make test‘ run with no error`		
Forbidden functions: `‘using namespace‘ keyword`		

Now that you have created your classes, it's time to use them! **NASA** has prepared some unit tests (in `RoverUnitTests.cpp`) to ensure that all the components are working properly, and that all errors are handled accordingly.

To run these tests, you are provided with the prototype files for each component of the **Rover**. As they are prototypes, the errors haven't been implemented and it's up to you to ensure that `make test` compiles and runs as expected.

All the files are in the subject

You can modify all files except `RoverUnitTest.cpp`

EXERCISE 2 - COMMUNICATION

	Exercise: 02	points : 3
Communication		
Turn-in directory: `cpp_d14a/ex02`		
Compiler: `g++`	Compilation flags: `-W -Wall -Wextra -Werror -std=c++14`	
Makefile: `No`	Rules: `n/a`	
Files to turn in: `Errors.hpp, Errors.cpp, CommunicationDevice.hpp, CommunicationDevice.cpp, CommunicationAPI.hpp, CommunicationAPI.cpp`		
Notes: `None`		
Forbidden functions: `‘using namespace‘ keyword`		

You now have to implement a `CommunicationDevice`. It will be used for communication between Houston and Mars.
You will have to use the `CommunicationAPI` and handle all its errors following these instructions:

- If `sendMessage` throws a standard exception, you should just print the error on the standard error output
- If `receiveMessage` throws a standard exception, you should also print the error on the standard error output, and the message should be *“INVALID MESSAGE”*
- If a standard exception is throw in `CommunicationDevice`’s constructor, you should catch it and throw a `CommunicationError` with the error preceded by *“Error:”* and a space (example: *“Error: userName should be at least 1 char.”*)
- The same goes for `startMission`, but with *“LogicError:”*, *“RuntimeError:”* and *“Error:”* as prefixes, for `std::logic_error`, `std::runtime_error` and `std::exception`, respectively

Exercise 3 - ScopedPtr

	Exercise: 03	points : 4
ScopedPtr		
Turn-in directory: cpp_d14a/ex03		
Compiler: g++	Compilation flags: -W -Wall -Wextra -Werror -std=c++14	
Makefile: No	Rules: n/a	
Files to turn in: SimplePtr.hpp, SimplePtr.cpp, BaseComponent.hpp, BaseComponent.cpp		
Notes: None		
Forbidden functions: ‘using namespace‘ keyword		

The aim of this exercise is to design a generic class to ensure the release of dynamically allocated components of the rover. For instance:

```
#include <stdexcept>

int main()
{
    try {
        // Use your auto delete here
        SimplePtr regulator(new AtmosphereRegulator);
        SimplePtr reclaimer(new WaterReclaimer);

        // The code above shouldn't be changed.
        throw std::runtime_error("An␣error␣occured␣here!");
    }
    catch (...) { }

    return 0;
}
```

SimplePtr.hpp is provided with the subject and doesn't need to be modified

Exercise 4 - RefPtr

KOALA	Exercise: 04	points : 5
RefPtr		
Turn-in directory: `cpp_d14a/ex04`		
Compiler: `g++`	Compilation flags: `-W -Wall -Wextra -Werror -std=c++14`	
Makefile: `No`	Rules: `n/a`	
Files to turn in: `RefPtr.hpp, RefPtr.cpp, BaseComponent.hpp, BaseComponent.cpp`		
Notes: `None`		
Forbidden functions: `‘using namespace‘ keyword`		

Our `ScopedPtr` is nice, but we can't copy it. Let's implement a `RefPtr` that can be stored, copied, and still takes care of deleting the object!

You are free to modify the provided class

Think of copy and assignment...

This code should construct a single `Oxygenator` and delete it.

```cpp
#include <stdexcept>
#include <cassert>

#include "RefPtr.hpp"
#include "Oxygenator.hpp"

int main()
{
    try {
        RefPtr oxygenator = new Oxygenator;
        BaseComponent *raw = oxygenator.get();
        RefPtr other(raw);
        RefPtr useless;
        RefPtr lastOne;
        lastOne = other;
        assert(lastOne.get() == raw);
        (void)useless;
        throw std::runtime_error("An error occured here!");
    }
    catch(...) { }

    return 0;
}
```