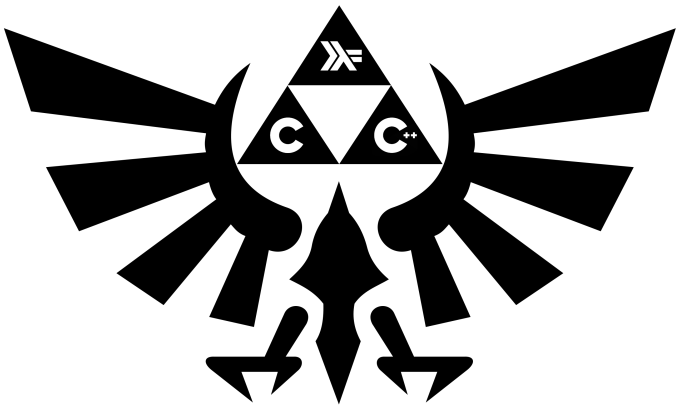
B3 - Paradigms Seminar

B-PDG-300

Day 10 PM

Everything is clear, except...



2.0

Day 10 PM

language: C++

- The totality of your source files, except all useless files (binary, temp files, obj files,...), must be included in your delivery.

All your exercises will be compiled with `g++` **and the** `-Wall -Wextra -Werror -std=c++20` **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 - ERRORS

Turn in: `Errors.hpp`, `Errors.cpp`
Notes: The `Errors.hpp` file is provided, but you must complete it.

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 must 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

Turn in: `Makefile`, `Errors.hpp/cpp`, `Engine.cpp/hpp`, `Oxygenator.hpp/cpp`
`AtmosphereRegulator.hpp/cpp`, `WaterReclaimer.hpp/cpp`
Provided: `BaseComponent.hpp`
Notes: Your objective is to have `make test` run with no error.

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

Turn in: `Errors.hpp/cpp`, `CommunicationDevice.hpp/cpp`, `CommunicationAPI.hpp/cpp`

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

Turn in: `SimplePtr.hpp/cpp`
** Provided**: `BaseComponent.hpp`

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

Turn in: `RefPtr.hpp/cpp`
Provided: `BaseComponent.hpp/cpp`

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.

```
#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;
}
```