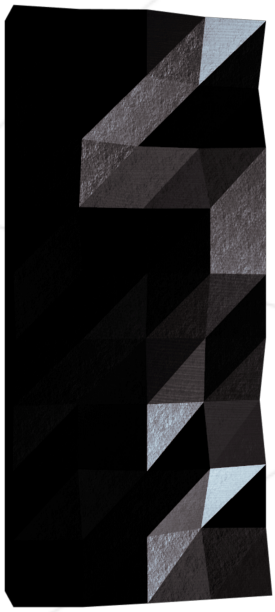

C++ Piscine - Module 05

Repetition and Exceptions

Summary: This document contains the subject for the module 05 of 42's C++ piscine.

Contents

I	General rules	2
II	Exercise 00: Mommy, when I grow up, I want to be a bureaucrat!	4
III	Exercise 01: Form up, maggots !	6
IV	Exercise 02: No, you need form 28B, not 28C ...	7
V	Exercise 03: At least this beats coffee-making	9
VI	Exercise 04: That's the way I like it, nice and boring	10
VII	Exercise 05: Endless red-tape generator	13

Chapter I

General rules

- Any function implemented in a header (except in the case of templates), and any unprotected header, means 0 to the exercise.
- Every output goes to the standard output, and will be ended by a newline, unless specified otherwise.
- The imposed filenames must be followed to the letter, as well as class names, function names and method names.
- Remember: You are coding in `C++` now, not in `C` anymore. Therefore:
  - The following functions are FORBIDDEN, and their use will be punished by a `0`, no questions asked: `*alloc`, `*printf` and `free`.
  - You are allowed to use basically everything in the standard library. HOWEVER, it would be smart to try and use the C++-ish versions of the functions you are used to in C, instead of just keeping to what you know, this is a new language after all. And NO, you are not allowed to use the STL until you actually are supposed to (that is, until module 08). That means no vectors/lists/maps/etc... or anything that requires an include <algorithm> until then.
- Actually, the use of any explicitly forbidden function or mechanic will be punished by a `0`, no questions asked.
- Also note that unless otherwise stated, the `C++` keywords `"using namespace"` and `"friend"` are forbidden. Their use will be punished by a `-42`, no questions asked.
- Files associated with a class will always be `ClassName.hpp` and `ClassName.cpp`, unless specified otherwise.
- Turn-in directories are `ex00/`, `ex01/`, ..., `exn/`.
- You must read the examples thoroughly. They can contain requirements that are not obvious in the exercise's description. If something seems ambiguous, you don't understand `C++` enough.
- Since you are allowed to use the `C++` tools you learned about since the beginning, you are not allowed to use any external library. And before you ask, that also means

no `C++11` and derivates, nor `Boost` or anything your awesomely skilled friend told you `C++` can't exist without.

- You may be required to turn in an important number of classes. This can seem tedious, unless you're able to script your favorite text editor.
- Read each exercise FULLY before starting it! Really, do it.
- The compiler to use is `clang++`.
- Your code has to be compiled with the following flags : `-Wall -Wextra -Werror`.
- Each of your includes must be able to be included independently from others. Includes must contains every other includes they are depending on, obviously.
- In case you're wondering, no coding style is enforced during in `C++`. You can use any style you like, no restrictions. But remember that a code your peer-evaluator can't read is a code she or he can't grade.
- Important stuff now : You will NOT be graded by a program, unless explictly stated in the subject. Therefore, you are afforded a certain amount of freedom in how you choose to do the exercises. However, be mindful of the constraints of each exercise, and DO NOT be lazy, you would miss a LOT of what they have to offer !
- It's not a problem to have some extraneous files in what you turn in, you may choose to separate your code in more files than what's asked of you. Feel free, as long as the result is not graded by a program.
- Even if the subject of an exercise is short, it's worth spending some time on it to be absolutely sure you understand what's expected of you, and that you did it in the best possible way.
- By Odin, by Thor! Use your brain!!!

Chapter II

Exercise 00: Mommy, when I grow up, I want to be a bureaucrat!

	Exercise : 00
Mommy, when I grow up, I want to be a bureaucrat!	
Turn-in directory : $ex00/$	
Files to turn in : `Bureaucrat.hpp Bureaucrat.cpp main.cpp`	
Forbidden functions : `None`	

Today, we're going to create an artificial nightmare of offices, corridors, forms, and waiting in line. Sounds fun ? No ? Too bad.

First, we start by the smallest cog in the vast bureaucratic machine : the `Bureaucrat`.

It must have a constant name, and a grade, that ranges from 1 (highest possible) to 150 (lowest possible). Any attempt to create a `Bureaucrat` with an invalid grade must throw an exception, which will be either a `Bureaucrat::GradeTooHighException` or a `Bureaucrat::GradeTooLowException`.

You will provide getters for both these attributes (`getName` and `getGrade`), and two functions to increment or decrement the grade. Both these functions will throw the same exceptions as before if the grade gets too high or too low. Remember, grade 1 is highest, 150 is lowest, so incrementing a grade 3 gives you a grade 2 ...

The exceptions you throw must be catchable by a block like :

```
try
{
        /* do some stuff with bureaucrats */
}
catch (std::exception & e)
{
        /* handle exception */
}
```

You will provide an overload of the << operator to `ostream` that outputs something like `<name>, bureaucrat grade <grade>`.

Of course, you will provide a `main` function to prove you did all this well.

Chapter III

Exercise 01: Form up, maggots !

	Exercise : 01
Form up, maggots !	
Turn-in directory : *ex*01/	
Files to turn in : `Same as before + Form.cpp Form.hpp`	
Forbidden functions : `None`	

Now that we have bureaucrats, better give them something to do with their time. What better occupation than a stack of forms to fill out ?

Make a `Form` class. It has a name, a boolean indicating whether it is signed (at the beginning, it's not), a grade required to sign it, and a grade required to execute it. The name and grades are constant, and all these attributes are private (not protected). The grades are subject to the same constraints as in the `Bureaucrat`, and exceptions will be thrown if any of them are out of bounds, `Form::GradeTooHighException` and `Form::GradeTooLowException`.

Same as before, make getters for all attributes, and an overload of the << operator to `ostream` that completely describes the state of the form.

You will also add a `beSigned` function that takes a `Bureaucrat`, and makes the form signed if the bureaucrat's grade is high enough. Always remember, grade 1 is better than grade 2. If the grade is too low, throw a `Form::GradeTooLowException`.

Also add a `signForm` function to the `Bureaucrat`. If the signing is successful, it will print something like `"<bureaucrat> signs <form>"`, otherwise it will print something like `"<bureaucrat> cannot sign <form> because <reason>"`.

Add whatever's needed to test this to your `main`.

Chapter IV

Exercise 02: No, you need form 28B, not 28C ...

	Exercise : 02
No, you need form 28B, not 28C ...	
Turn-in directory : *ex*02/	
Files to turn in : `Same as before + ShrubberyCreationForm.[hpp,cpp] RobotomyRequestForm.[hpp,cpp] PresidentialPardonForm.[hpp,cpp]`	
Forbidden functions : `None`	

Now that you have basic forms, you will make a few forms that actually do something.

Create a few concrete forms :

- `ShrubberyCreationForm` (Required grades : sign 145, exec 137). Action : Create a file called <target>_shrubbery, and write ASCII trees inside it, in the current directory.
- `RobotomyRequestForm` (Required grades : sign 72, exec 45). Action : Makes some drilling noises, and tells us that <target> has been robotomized successfully 50% of the time. the rest of times, tells us it's a failure.
- `PresidentialPardonForm` (Required grades : sign 25, exec 5). Action : Tells us <target> has been pardoned by Zafod Beeblebrox.

All of these will have to take only one parameter in their constructor, which will represent the target of the form. For example, "home" if you want to plant a shrubbery at home. Remember the form's attributes need to remain private, and in the base class.

Now you need to add a method `execute(Bureaucrat const & executor) const` to the base form, and implement a method actually executing the form's action in all the concrete forms. You have to check that the form is signed, and that the bureaucrat attempting to execute the form has a high enough grade, else you will throw an appropriate exception. Whether you want to make these checks in every concrete class or make the

check in the base class then calling another method to actually execute the action is up to you, but one way is obviously prettier than the other one. In any event, the base form must be an abstract class.

Finish this by adding an `executeForm(Form const & form)` function to the bureaucrat. It must attempt to execute the form, and if it's a success, print something like `<bureaucrat> executes <form>`. If not, print an explicit error message.

Add whatever you need to your `main` to test that everything works.

Chapter V

Exercise 03: At least this beats coffee-making

	Exercise : 03
At least this beats coffee-making	
Turn-in directory : *ex*03/	
Files to turn in : `Same as before + Intern.hpp Intern.cpp`	
Forbidden functions : `None`	

Because filling out forms is annoying enough, it would just be cruel to ask our bureaucrats to write them entirely by themselves. No, we'll just have an intern do that.

You're going to create the `Intern` class. The intern has no name, no grade, no defining characteristics whatsoever, we only care that it does its job.

The intern has one important thing, the `makeForm` function. It takes two strings, the first representing the name of a form, and the second one being the target for the form. It will return, as a pointer to `Form`, a pointer to whichever concrete form class is represented by the first parameter, initialized with the second parameter. It will print something like `"Intern creates <form>"` to the standard output. If the requested form is not known, print an explicit error message.

For example, this would create a `RobotomyRequestForm` targeted on `"Bender"` :

```
{
        Intern  someRandomIntern;
        Form*   rrf;

        rrf = someRandomIntern.makeForm("robotomy request", "Bender");
}
```

Your `main` must, of course, test all of this.

Chapter VI

Exercise 04: That's the way I like it, nice and boring

	Exercise : 04
That's the way I like it, nice and boring	
Turn-in directory : *ex*04/	
Files to turn in : `Same as before + OfficeBlock.cpp OfficeBlock.hpp`	
Forbidden functions : `None`	

`This exercise and the following ones do not offer any points but are still useful for your piscine. You can do them, or not.`

The Central Bureaucracy, being the haven of order and organization that it is, is constituted of cleanly arranged office blocks. Each of these blocks requires an intern and two bureaucrats to function, and is capable of creating, signing and executing forms, all by just giving it an order. Cool, isn't it ?

So, make an `OfficeBlock` class. It will be constructed by passing pointers to (or references to, you decide depending on what's appropriate) one intern, one signing bureaucrat and one executing bureaucrat. It can also be constructed empty. No other construction must be possible (No copy, no assignation).

It will have functions to set a new intern, signing bureaucrat, or executing bureaucrat.

Its only "useful" function will be `doBureaucracy`, it takes a form name and a target name. It will attempt to, in order, make the intern create the requested form, have the second bureaucrat sign it, and have the second bureaucrat execute it. The messages printed by the intern and bureaucrats will provide a log of what's happening. When an error occurs, it must result in an exception being raised from this function : You are

free to modify what you did before to make this elegant. Remember : Specific errors are always cool.

Of course, if not all three spots in the block are filled, no bureaucracy can be done.

As usual, your `main` must be ... well you know the drill.

For example, the following block of code may produce the following output:

```
int main()
{
        Intern      idiotOne;
        Bureaucrat  hermes = Bureaucrat("Hermes Conrad", 37);
        Bureaucrat  bob = Bureaucrat("Bobby Bobson", 123);
        OfficeBlock ob;

        ob.setIntern(idiotOne);
        ob.setSigner(bob);
        ob.setExecutor(hermes);

        try
        {
                ob.doBureaucracy("mutant pig termination", "Pigley");
        }
        catch (Some::SpecificException & e)
        {
                /* specific known error happens, say something */
        }
        catch (std::exception & e)
        {
                /* oh god, unknown error, what to do ?! */
        }
}
```

```
$> ./ex04
Intern creates a Mutant Pig Termination Form (s.grade 130, ex.grade 50) targeted on Pigley (Unsigned)
Bureaucrat Bobby Bobson (Grade 123) signs a Mutant Pig Termination Form (s.grade 130, ex.grade 50)
    targeted on Pigley (Unsigned)
Bureaucrat Hermes Conrad (Grade 37) executes a Mutant Pig Termination Form (s.grade 130, ex.grade 50)
    targeted on Pigley (Signed)
That'll do, Pigley. That'll do ...
$>
```

Chapter VII

Exercise 05: Endless red-tape generator

	Exercise : 05
Endless red-tape generator	
Turn-in directory : $ex05/$	
Files to turn in : `Same as before + CentralBureaucracy.cpp CentralBureaucracy.hpp`	
Forbidden functions : `None`	

Now you just have to wrap this all up into a neat little package.

Create the `CentralBureaucracy` class. It will be created without parameters, and at its creation will have 20 empty office blocks.

It will be possible to "feed" bureaucrats to the object. Interns will be generated automatically, without user intervention, because let's face it, interns are a dime a dozen.

Bureaucrats that are fed to the object will be used to fill seats in its office blocks. If no seats are available, you can either reject them, or store them in a waiting area somewhere.

After that, it will be possible to queue targets in the object, using a `queueUp` function that takes a string, the name of the person in queue.

Finally, when a `doBureaucracy` function is called, some random bit of bureaucracy will be done to each one of them, first arrived first served, by each office block in sequence.

So, here's what your main function might look like :

- Create the Central Bureaucracy
- Create 20 random bureaucrats and feed them to the Central Bureaucracy
- Queue up a large number of targets in the Central Bureaucracy

- Call the doBureaucracy function and watch magic happen