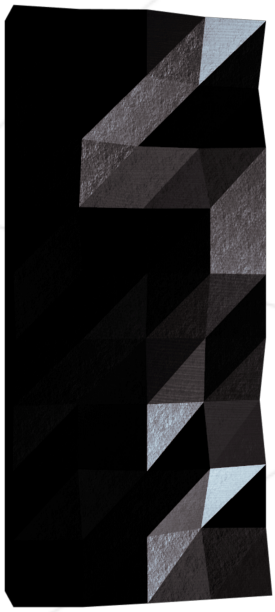

C++ - Module 05

Repetition and Exceptions

Summary:
This document contains the exercises of Module 05 from C++ modules.

Version: 9

Contents

Chapter I

Introduction

C++ is a general-purpose programming language created by Bjarne Stroustrup as an extension of the C programming language, or "C with Classes" (source: Wikipedia).

The goal of these modules is to introduce you to **Object-Oriented Programming**. This will be the starting point of your C++ journey. Many languages are recommended to learn OOP. We decided to choose C++ since it's derived from your old friend C. Because this is a complex language, and in order to keep things simple, your code will comply with the C++98 standard.

We are aware modern C++ is way different in a lot of aspects. So if you want to become a proficient C++ developer, it's up to you to go further after the 42 Common Core!

Chapter II

General rules

Compiling

- Compile your code with `c++` and the flags `-Wall -Wextra -Werror`
- Your code should still compile if you add the flag `-std=c++98`

Formatting and naming conventions

- The exercise directories will be named this way: `ex00, ex01, ... , exn`
- Name your files, classes, functions, member functions and attributes as required in the guidelines.
- Write class names in **UpperCamelCase** format. Files containing class code will always be named according to the class name. For instance: `ClassName.hpp`/`ClassName.h`, `ClassName.cpp`, or `ClassName.tpp`. Then, if you have a header file containing the definition of a class "BrickWall" standing for a brick wall, its name will be `BrickWall.hpp`.
- Unless specified otherwise, every output messages must be ended by a new-line character and displayed to the standard output.
- *Goodbye Norminette!* No coding style is enforced in the C++ modules. You can follow your favorite one. But keep in mind that a code your peer-evaluators can't understand is a code they can't grade. Do your best to write a clean and readable code.

Allowed/Forbidden

You are not coding in C anymore. Time to C++! Therefore:

- You are allowed to use almost everything from the standard library. Thus, instead of sticking to what you already know, it would be smart to use as much as possible the C++-ish versions of the C functions you are used to.
- However, you can't use any other external library. It means C++11 (and derived forms) and `Boost` libraries are forbidden. The following functions are forbidden too: `*printf()`, `*alloc()` and `free()`. If you use them, your grade will be 0 and that's it.

- Note that unless explicitly stated otherwise, the `using namespace <ns_name>` and `friend` keywords are forbidden. Otherwise, your grade will be -42.
- **You are allowed to use the STL in Module 08 only.** That means: no **Containers** (vector/list/map/and so forth) and no **Algorithms** (anything that requires to include the `<algorithm>` header) until then. Otherwise, your grade will be -42.

A few design requirements

- Memory leakage occurs in C++ too. When you allocate memory (by using the `new` keyword), you must avoid **memory leaks**.
- From Module 02 to Module 08, your classes must be designed in the **Orthodox Canonical Form, except when explicitely stated otherwise**.
- Any function implementation put in a header file (except for function templates) means 0 to the exercise.
- You should be able to use each of your headers independently from others. Thus, they must include all the dependencies they need. However, you must avoid the problem of double inclusion by adding **include guards**. Otherwise, your grade will be 0.

Read me

- You can add some additional files if you need to (i.e., to split your code). As these assignments are not verified by a program, feel free to do so as long as you turn in the mandatory files.
- Sometimes, the guidelines of an exercise look short but the examples can show requirements that are not explicitly written in the instructions.
- Read each module completely before starting! Really, do it.
- By Odin, by Thor! Use your brain!!!

You will have to implement a lot of classes. This can seem tedious, unless you're able to script your favorite text editor.

You are given a certain amount of freedom to complete the exercises. However, follow the mandatory rules and don't be lazy. You would miss a lot of useful information! Do not hesitate to read about theoretical concepts.

Chapter III

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 : *ex*00/	
Files to turn in : `Makefile, main.cpp, Bureaucrat.{h, hpp}, Bureaucrat.cpp`	
Forbidden functions : `None`	

`Please note that exception classes don't have to be designed in Orthodox Canonical Form. But every other classes have to.`

Let's design an artificial nightmare of offices, corridors, forms, and waiting queues. Sounds fun? No? Too bad.

First, start by the smallest cog in this vast bureaucratic machine: the **Bureaucrat**.

A **Bureaucrat** must have:

- A constant name.
- And a grade that ranges from **1** (highest possible grade) to **150** (lowest possible grade).

Any attempt to instantiate a Bureaucrat using an invalid grade must throw an exception:
either a `Bureaucrat::GradeTooHighException` or a `Bureaucrat::GradeTooLowException`.

You will provide getters for both these attributes: `getName()` and `getGrade()`. Implement also two member functions to increment or decrement the bureaucrat grade. If the grade is out of range, both of them will throw the same exceptions as the constructor.

Remember. Since grade 1 is the highest one and 150 the lowest, incrementing a grade 3 should give a grade 2 to the bureaucrat.

The thrown exceptions must be catchable using try and catch blocks:

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

You will implement an overload of the insertion («) operator to print something like (without the angle brackets):

`<name>, bureaucrat grade <grade>.`

As usual, turn in some tests to prove everything works as expected.

Chapter IV

Exercise 01: Form up, maggots!

	Exercise : 01
Form up, maggots!	
Turn-in directory : $ex01/$	
Files to turn in : `Files from previous exercise + Form.{h, hpp}, Form.cpp`	
Forbidden functions : `None`	

Now that you have bureaucrats, let's give them something to do. What better activity could there be than the one of filling out a stack of forms?

Then, let's make a **Form** class. It has:

- A constant name.
- A boolean indicating whether it is signed (at construction, it's not).
- A constant grade required to sign it.
- And a constant grade required to execute it.

All these attributes are **private**, not protected.

The grades of the **Form** follow the same rules that apply to the Bureaucrat. Thus, the following exceptions will be thrown if a form grade is out of bounds:
`Form::GradeTooHighException` and `Form::GradeTooLowException`.

Same as before, write getters for all attributes and an overload of the insertion («) operator that prints all the form's informations.

Add also a `beSigned` member function to the Form that takes a Bureaucrat as parameter. It changes the form status to signed if the bureaucrat's grade is high enough (higher or egal to the required one). Remember, grade 1 is higher than grade 2. If the grade is too low, throw a `Form::GradeTooLowException`.

Lastly, add a `signForm` member function to the Bureaucrat. If the form got signed, it will print something like:

`<bureaucrat> signed <form>`

Otherwise, it will print something like:

`<bureaucrat> couldn't sign <form> because <reason>`.

Implement and turn in tests to ensure everything works as expected.

Chapter V

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

	Exercise : 02
No, you need form 28B, not 28C...	
Turn-in directory : $ex02/$	
Files to turn in : `Files from previous exercises + ShrubberyCreationForm.[{h, hpp},cpp], RobotomyRequestForm.[{h, hpp},cpp], PresidentialPardonForm.[{h, hpp},cpp]`	
Forbidden functions : `None`	

Since you now have basic forms, it's time to make a few more that actually do something.
In all cases, the base class Form must be an abstract class. Keep in mind the form's attributes need to remain private and that they are in the base class.

Add the following concrete classes:

- **ShrubberyCreationForm**: Required grades: sign 145, exec 137
  Create a file `<target>_shrubbery` in the working directory, and writes ASCII trees inside it.
- **RobotomyRequestForm**: Required grades: sign 72, exec 45
  Makes some drilling noises. Then, informs that `<target>` has been robotomized successfully 50% of the time. Otherwise, informs that the robotomy failed.
- **PresidentialPardonForm**: Required grades: sign 25, exec 5
  Informs that `<target>` has been pardoned by Zaphod Beeblebrox.

All of them take only one parameter in their constructor: the target of the form. For example, "home" if you want to plant shrubbery at home.

Now, add the `execute(Bureaucrat const & executor) const` member function to the base form and implement a function to execute the form's action of the concrete classes. You have to check that the form is signed and that the grade of the bureaucrat attempting to execute the form is high enough. Otherwise, throw an appropriate exception.

Whether you want to check the requirements in every concrete class or in the base class (then call another function to execute the form) is up to you. However, one way is prettier than the other one.

Lastly, add the `executeForm(Form const & form)` member function to the Bureaucrat. It must attempt to execute the form. If it's successful, print something like:

```
<bureaucrat> executed <form>
```

If not, print an explicit error message.

Implement and turn in tests to ensure everything works as expected.

Chapter VI

Exercise 03: At least this beats coffee-making

	Exercise : 03
At least this beats coffee-making	
Turn-in directory : $ex03/$	
Files to turn in : `Files from previous exercises + Intern.{h, hpp}, Intern.cpp`	
Forbidden functions : `None`	

Because filling out forms is annoying enough, it would be cruel to ask our bureaucrats to do this all day long. Fortunately, interns exist. In this exercise, you have to implement the **Intern** class. The intern has no name, no grade, no unique characteristics. The only thing the bureaucrats care about is that they do their job.

However, the intern has one important capacity: the `makeForm()` function. It takes two strings. The first one is the name of a form and the second one is the target of the form. It return a pointer to a **Form object** (whose name is the one passed as parameter) whose target will be initialized to the second parameter.

It will print something like:

```
Intern creates <form>
```

If the form name passed as parameter doesn't exist, print an explicit error message.

You must avoid unreadable and ugly solutions like using a if/elseif/else forest. This kind of things won't be accepted during the evaluation process. You're not in Piscine (pool) anymore. As usual, you have to test that everything works as expected.

For example, the code below creates a **RobotomyRequestForm** targeted on "Bender":

```
{
    Intern  someRandomIntern;
    Form*   rrf;

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