C++ - Module 05

Repetition and Exceptions

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

Version: 10.3

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 for learning OOP. We have chosen C++ since it is 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 that modern C++ is significantly different in many aspects. So, if you want to become a proficient C++ developer, it is 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 message must end with a newline character and be 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 code your peer evaluators can't understand is code they can't grade. Do your best to write 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 the C++-ish versions of the C functions you are used to as much as possible.
- 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 only in Modules 08 and 09.** That means: no **Containers** (vector/list/map, and so forth) and no **Algorithms** (anything that requires including 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 09, your classes must be designed in the **Orthodox Canonical Form, except when explicitly 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!!!

Regarding the Makefile for C++ projects, the same rules as in C apply (see the Norm chapter about the Makefile).

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 do not have to be designed in Orthodox Canonical Form. However, every other class must follow it.`

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

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

A **Bureaucrat** must have:

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

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

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

Remember, since grade 1 is the highest and 150 the lowest, incrementing a grade 3 should result in a grade 2 for 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 must implement an overload of the insertion («) operator to print output in the following format (without the angle brackets):

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

As usual, submit some tests to prove that 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 the 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 filling out a stack of forms?

Let's create a **Form** class. It has:

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

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

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

As before, write getters for all attributes and overload the insertion («) operator to print all the form's information.

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

Then, modify the `signForm()` member function in the Bureaucrat class. This function must call `Form::beSigned()` to attempt to sign the form. If the form is signed successfully, it will print something like:
`<bureaucrat> signed <form>`

Otherwise, it will print something like:

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

Implement and submit some 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 : `Makefile, main.cpp, Bureaucrat.[{h, hpp},cpp], + AForm.[{h, hpp},cpp], ShrubberyCreationForm.[{h, hpp},cpp], + RobotomyRequestForm.[{h, hpp},cpp], PresidentialPardonForm.[{h, hpp},cpp]`	
Forbidden functions : `None`	

Now that you have basic forms, it's time to create a few more that actually do something.

In all cases, the base class Form must be an abstract class and should therefore be renamed AForm. Keep in mind that the form's attributes need to remain private and that they belong to the base class.

Add the following concrete classes:

- **ShrubberyCreationForm**: Required grades: sign 145, exec 137
  Creates 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, it 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 in the concrete classes. You must 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 check the requirements in every concrete class or in the base class (and then call another function to execute the form) is up to you. However, one way is more elegant than the other.

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

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

If not, print an explicit error message.

Implement and submit some 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 : *ex*03/	
Files to turn in : `Files from previous exercises + Intern.{h, hpp}, Intern.cpp`	
Forbidden functions : `None`	

Since filling out forms all day would be too cruel for our bureaucrats, interns exist to take on this tedious task. In this exercise, you must implement the **Intern** class. The intern has no name, no grade, and no unique characteristics. The only thing bureaucrats care about is that they do their job.

However, the intern has one key ability: the `makeForm()` function. This function takes two strings as parameters: the first one represents the name of a form, and the second one represents the target of the form. It returns a pointer to a **AForm** object (corresponding to the form name passed as a parameter), with its target initialized to the second parameter.

It should print something like:

```
Intern creates <form>
```

If the provided form name does not exist, print an explicit error message.

You must avoid unreadable and messy solutions, such as using an excessive if/elseif/else structure. This kind of approach will not be accepted during the evaluation process. You're not in the Piscine (pool) anymore. As usual, you must test everything to ensure it works as expected.

For example, the following code creates a **RobotomyRequestForm** targeted at "Bender":

```
{
    Intern  someRandomIntern;
    AForm*   rrf;

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

Chapter VII

Submission and Peer Evaluation

Submit your assignment to your `Git` repository as usual. Only the work inside your repository will be evaluated during the defense. Make sure to double-check the names of your folders and files to ensure they are correct.

16D85ACC441674FBA2DF65190663F9373230CEAB1E4A0818611C0E39F5B26E4D774F1
74620A16827E1B16612137E59ECD492E468A92DCB17BF16988114B98587594D12810
E67D173222A