B3 - C++ Pool

B-CPP-300

Day 06

IOStream, String and objects

2.1

Day 06

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` **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 at the root of your repository unless specified otherwise**.

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

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.

For them to be executed and evaluated, put a `Makefile` at the root of your directory with the `tests_run` rule as mentionned in the documentation linked above.
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 - IOStream

Turn in: `Makefile` and your program files in `ex00/`
Makefile rules: `all`, `clean`, `fclean`, `re`
Notes: You must turn in your complete program, including your `main` function.

Your `Makefile` must generate a `my_cat` executable.

Write a simplified `cat(1)` command.
Your executable must take one or several files as parameters, and does not need to handle the special case of the standard input.
Upon error (file not found, permission denied, etc.), you must write the following message to the error output:

```
my_cat: file: No such file or directory
```

`file` must be replaced with the name of the file for which the error was encountered.

If no parameter is passed to your program, you must write the following message to the error output:

```
my_cat: Usage: ./my_cat file [...]
```

Exercise 1 - Temperature Conversion

Turn in: `Makefile` and your program files in `ex01/`
Makefile rules: `all`, `clean`, `fclean`, `re`
Notes: You must turn in your complete program, including your `main` function

Your `Makefile` must generate a `my_convert_temp` executable.

The purpose of this exercise is to write a program that converts temperatures from the `Celsius` scale to the `Fahrenheit` scale, and vice-versa.
The conversion formula to use is the following (we know, it isn't the exact right one!):

```
Celsius = 5.0 / 9.0 * ( Fahrenheit - 32 )
```

Your program must read from its standard input (separated by one or more spaces):

- a temperature
- a scale

Any additional input must be ignored.

Terminal

```
~/B-CPP-300> ./my_convert_temp << EOF
-10 Celsius
EOF
          14.000      Fahrenheit
~/B-CPP-300> ./my_convert_temp << EOF
46.400 Fahrenheit
EOF
           8.000         Celsius
```

Results must be displayed within two columns, right-aligned with a padding of 16 composed of spaces and a precision to the 1000th.

Exercise 2 - The Patient

Turn in: `SickKoala.hpp`, `SickKoala.cpp` in `hospital/`

You are now working on a simulation of your dear Koalas' health.
To get started, you'll need patients to treat.
Therefore, it is time to **create a** `SickKoala` **class**.
Here are the information you need to implement this class:

- they can't be instantiated without a name `string`,
- following their destruction, the standard output must display

```
Mr.[name]: Kreoooogg!! I'm cuuuured!
```

- a `poke` member function taking no parameters or return value and displaying the following when called:

```
Mr.[name]: Gooeeeeerrk!!
```

- a `takeDrug` function taking a `string` as parameter and returning `true` if the `string` matches `Mars` or `Kinder` (case sensitive).
  The function must then display, respectively:

```
Mr.[name]: Mars, and it kreogs!
```

  or

```
Mr.[name]: There is a toy inside!
```

  In any other case, the function returns `false` and displays:

```
Mr.[name]: Goerkreog!
```

- sometimes, `SickKoalas` go crazy when their fever is too high.
  To simulate this, `SickKoalas` have an `overDrive` member function that returns nothing and takes a `string` as parameter.
  It displays the string passed as parameter, preceded by `"Mr.[name]: "`, within which all occurences of `"Kreog!"` are replaced by `"1337!"`.
  For instance:

```
Kreog! How's it going?
```

  becomes:

```
Mr.[name]: 1337! How's it going?
```

For all outputs in this exercise, `[name]` must be replaced by the name of the `SickKoala`

Exercise 3 - The Nurse

Turn in: `KoalaNurse.hpp`, `KoalaNurse.cpp`, `SickKoala.hpp`, `SickKoala.cpp` in `hospital/`

Now that we have patients, we need a nurse to take care of them.
You are now coding the nurse for the koala: **the `KoalaNurse` class**.
Here is the information you need in order to create the `KoalaNurse`:

- each `KoalaNurse` has a numerical identifier (ID) which must be provided when the object is created, but it is not possible to create a nurse without specifying her ID,
- when a `KoalaNurse` is destroyed, it'll express its relief like so:

```
Nurse [ID]: Finally some rest!
```

- the nurse can give drugs to patients, through a `giveDrug` member function with the following parameters a `string` (Drug) and a pointer to the patient.
  This member function does not return anything.
  When it is called, the nurse gives medication to the patient.
- the nurse can read the doctor's report through a `readReport` member function that takes a filename `string` as parameter.
  - The filename is built from the sick Koala's name, followed by the `.report` extension.
  - The file contains the name of the drug to give to the patient.

This member function returns the name of the drug as a `string` and prints the following to the standard output:

```
Nurse [ID]: Kreog! Mr.[patientName] needs a [drugName]!
```

If the `.report` file doesn't exist or is not valid, nothing must be displayed and the return value must be an empty `string`.

- the nurse can clock in thanks to a `timeCheck` member function that takes no parameter and doesn't return anything
  The nurse calls this member function when it starts working and when it stops working (as it is a very diligent worker).
  When it clocks in at the start of her job, it says:

```
Nurse [ID]: Time to get to work!
```

When it stops working, it says:

```
Nurse [ID]: Time to go home to my eucalyptus forest!
```

It is up to you to figure out a way to find out when it starts and stops working.

By default, when the program starts, the nurse is not working yet.
The `KoalaNurse` being very diligent, it will take any job.
Even outsided the hospital.
Only a call to the `timeCheck` member function lets the `KoalaNurse` change her working status: if it is not working, it starts to work; if it is working, it stops.

In this exercise, `[ID]` must be replaced with the `KoalaNurse`'s ID in any output

Exercise 4 - The Doctor

Turn in: `KoalaDoctor.hpp/cpp`, `KoalaNurse.hpp/cpp`, `SickKoala.hpp/cpp` in `hospital/`

Before we get started, add a `getName` member function to the `SickKoala` class, taking no parameters and returning the name of the patient as a `string`.

We now have patients and nurses taking care of them.
We still need a doctor to give instructions to the nurses.
Implement a simulation of the doctor with the `KoalaDoctor` class.

Here's what we know about the `KoalaDoctor`:

- it must be instantiated with a name `string`.
  During construction, it must print the following to the standard output:

```
Dr.[name]: I'm Dr.[name]! How do you kreog?
```

- it can diagnose patients using the `diagnose` member function that takes a pointer to the patient to diagnose as parameter.
  This member function prints the following to the standard output:

```
Dr.[name]: So what's goerking you Mr.[patientName]?
```

It then calls the `poke` member function of the `SickKoala`.
The doctor then writes a report for nurses, in a file named `[patientname].report`.
This file contains the name of the drug to give to the patient. The name will be picked at random from the following list:

- Mars
- Kinder
- Crunch
- Maroilles
- Eucalyptus leaf

To do this, you must use `random()% 5` on the previous list, in the given order.
The `srandom` function will be called by the correction `main`.

- the `KoalaDoctor` clocks in through a `timeCheck` member function, which takes no parameters and does not return anything.
  When it starts working, it says:

```
Dr.[name]: Time to get to work!
```

When it stops working, it says:

```
Dr.[name]: Time to go home to my eucalyptus forest!
```

The `KoalaDoctor` being very diligent, it will take any job.
Even outside the hospital.

In this exercise, any occurence of `[name]` must be replaced with the name of the `KoalaDoctor`, and occurences of `[patientName]` must be replaced with the name of the `SickKoala` that is currently being treated.

Exercise 5 - Lists

Turn in: `KoalaDoctor.hpp/cpp`, `KoalaNurse.hpp/cpp`, `SickKoala.hpp/cpp`, `SickKoalaList.hpp/cpp`, `KoalaNurseList.hpp/cpp`, `KoalaDoctorList.hpp/cpp` in `hospital/`
Notes: Recursive programming can save you a lot of development time…

Before we get started, modify your `KoalaNurse` and `KoalaDoctor` classes:

- Add a `getID` member function to the `KoalaNurse` class.
  This function takes no parameter and returns an `int`.
- Add a `getName` member function to the `KoalaDoctor` class.
  This function takes no parameter and returns a `string`.

We now need to watch over all these people working together in harmony.
It is necessary to be able to handle several patients, doctors and/or nurses at the same time.
To do so, we need a list for each of these categories.

For this exercise, a node of a list is a `List *` object.

Implement the 3 following classes.

SickKoalaList

- takes a pointer to a `SickKoala` as a constructor parameter.
  This pointer can be a `nullptr`.
- has an `isEnd` member function which takes no parameter and returns a boolean set to `true` if the `SickKoalaList` is the last node of its list.
- has an `append` member function which takes a pointer to a `SickKoalaList` as a parameter and does not return anything.
  The node passed as parameter is added to the end of the linked list.
- has a `getFromName` member function which takes a `string` as a parameter and returns a pointer to the first `SickKoala` in the list whose name matches that `string`.
- has a `remove` member function which takes a pointer to a `SickKoalaList` as a parameter and removes the `SickKoalaList` matching this pointer from the list.
  It returns a pointer to the first node of the list.
- has a `removeFromName` member function which takes a `string` as a parameter and removes the first `SickKoalaList` whose content's name matches that `string` from the list.
  It returns a pointer to the first node of the list.
- has a `getContent` member function which takes no parameter and returns a pointer to the element held in the current instance.

- has a `getNext` member function which takes no parameter and returns a pointer to the next node of the list.
  If there is no such node, the function returns a `nullptr`.
- has a `dump` member function which takes no parameter and does not return anything.
  It displays the name of all the `SickKoalas` in the list in order (begin -> end):

```
Patients: name1, name2, ..., nameX.
```

If an element is missing, the name to display is `[nullptr]`.

KoalaNurseList

- takes a pointer to a `KoalaNurse` as a constructor parameter.
  This pointer can be a `nullptr`.
- has an `isEnd` member function which takes no parameter and returns a boolean set to `true` if the `KoalaNurseList` is the last node of its list.
- has an `append` member function which takes a pointer to a `KoalaNurseList` as a parameter and does not return anything.
  The node passed as parameter is added to the end of the linked list.
- has a `getFromID` member function which takes an `int` as a parameter and returns a pointer to the first `KoalaNurse` in the list whose ID matches that `int`.
- has a `remove` member function which takes a pointer to a `KoalaNurseList` and removes the `KoalaNurseList` matching this pointer from the list.
  It returns a pointer to the first node of the list.
- has a `removeFromID` member function which takes an `int` as parameter and removes the first `KoalaNurseList` whose content's ID matches that `int` from the list.
  It returns a pointer to the first node of the list.
- has a `dump` member function which takes no parameter and does not return anything.
  It displays the ID of all the `KoalaNurses` in the list in order (begin -> end):

```
Nurses: id1, id2, ..., idX.
```

If an element is missing, the ID to display is `[nullptr]`.

KoalaDoctorList

- takes a pointer to a `KoalaDoctor` as a constructor parameter.
  This pointer can be a `nullptr`.
- has an `isEnd` member function which takes no parameter and returns a boolean set to `true` if the `KoalaDoctorList` is the last node of its list.

- has an `append` member function which takes a pointer to a `KoalaDoctorList` as a parameter and does not return anything. The node passed as parameter is added to the end of the linked list.
- has a `getFromName` member function which takes a `string` as a parameter and returns the first `KoalaDoctor` in the list whose name matches that `string`.
- has a `remove` member function which takes a pointer to a `KoalaDoctorList` as a parameter and removes the `KoalaDoctorList` matching this pointer from the list.
  It returns a pointer to the first node of the list.
- has a `removeFromName` member function which takes a `string` as a parameter and removes the first `KoalaDoctorLis` whose content's name matches that `string` from the list.
  It returns a pointer to the first node of the list.
- has a `dump` member function which takes no parameter and does not return anything.
  It displays the name of all `KoalaDoctors` in the list in order (begin -> end):

```
Doctors: name1, name2, ..., nameX.
```

If an element is missing, the name to display is `[nullptr]`.

Exercise 6 - The Hospital

Turn in: `KoalaDoctor.hpp/cpp`, `KoalaNurse.hpp/cpp`, `SickKoala.hpp/cpp`, `SickKoalaList.hpp/cpp`, `KoalaNurseList.hpp/cpp`, `KoalaDoctorList.hpp/cpp`, `Hospital.hpp/cpp` in `hospital/`

It is now possible to manage several patients, nurses and doctors.
It is time to move on and manage the entire `Hospital`!
You will now code without any help.
You must deduce the member functions of the `Hospital` based on the sample `main` function you will find below.

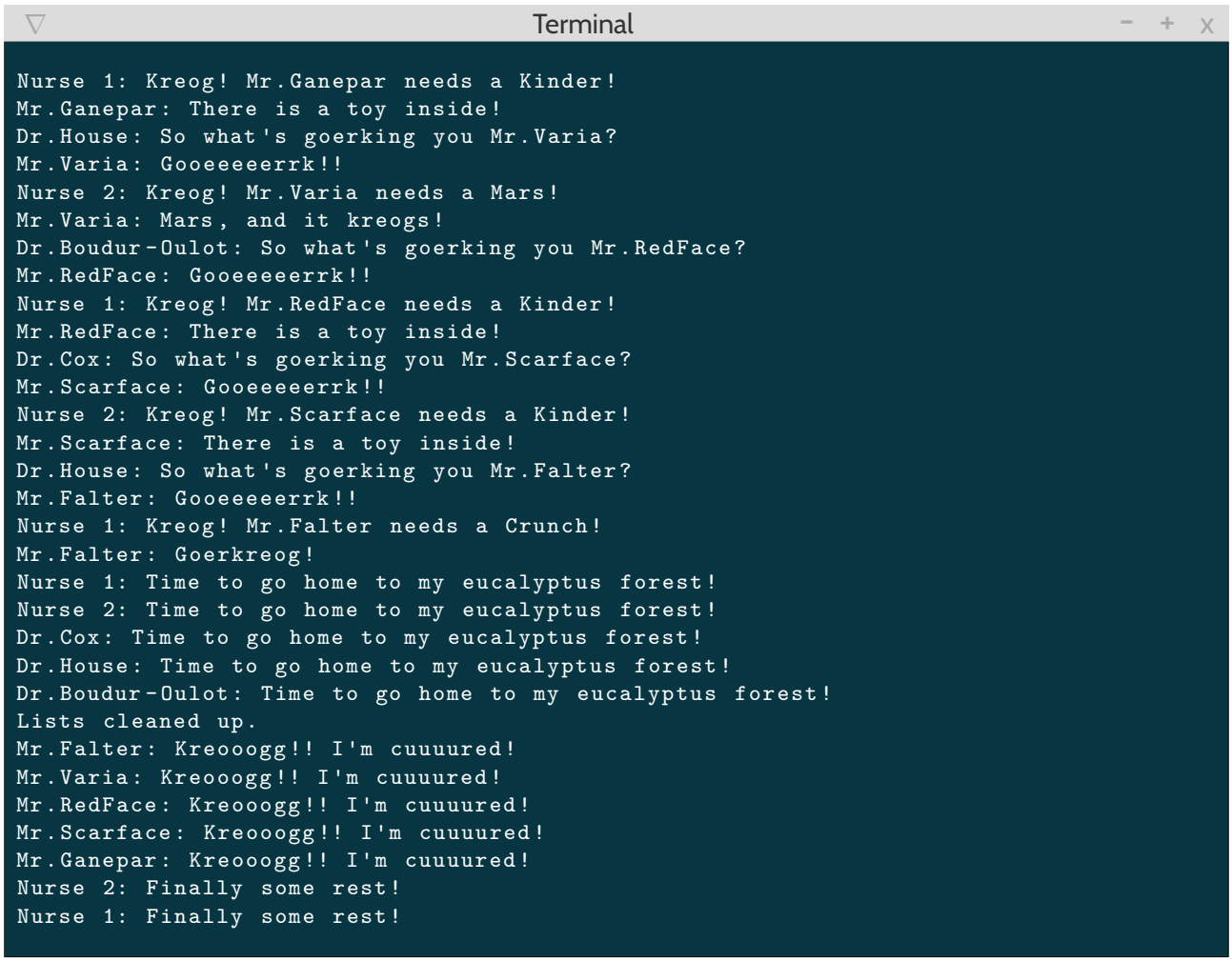
The `Hospital` must distribute work between doctors and nurses.
For this exercise, you may have to modify existing classes.
You are responsible for these modifications, as long as they comply with the requirements and descriptions of the previous exercises!

```
#include <iostream>
#include <string>
#include <cstdlib>
#include "SickKoala.hpp"
#include "KoalaNurse.hpp"
#include "KoalaDoctor.hpp"
#include "SickKoalaList.hpp"
#include "KoalaNurseList.hpp"
#include "KoalaDoctorList.hpp"
#include "Hospital.hpp"

int main(void)
{
    srandom(42);

    KoalaDoctor     cox("Cox");
    KoalaDoctor     house("House");
    KoalaDoctor     tired("Boudur-Oulot");
    KoalaDoctorList doc1(&cox);
    KoalaDoctorList doc2(&house);
    KoalaDoctorList doc3(&tired);

    KoalaNurse      ratched(1);
    KoalaNurse      betty(2);
    KoalaNurseList  nurse1(&ratched);
    KoalaNurseList  nurse2(&betty);

    SickKoala       cancer("Ganepar");
    SickKoala       gangrene("Scarface");
    SickKoala       measles("RedFace");
    SickKoala       smallpox("Varia");
    SickKoala       fracture("Falter");
    SickKoalaList   sick1(&cancer);
    SickKoalaList   sick4(&gangrene);
    SickKoalaList   sick3(&measles);
    SickKoalaList   sick2(&smallpox);
    SickKoalaList   sick5(&fracture);
```

```
    {
        Hospital bellevue;

        bellevue.addDoctor(&doc1);
        bellevue.addDoctor(&doc2);
        bellevue.addDoctor(&doc3);
        bellevue.addSick(&sick1);
        bellevue.addSick(&sick2);
        bellevue.addSick(&sick3);
        bellevue.addSick(&sick4);
        bellevue.addSick(&sick5);
        bellevue.addNurse(&nurse1);
        bellevue.addNurse(&nurse2);
        bellevue.addSick(&sick4);
        bellevue.run();
    }

    if( nurse1.isEnd() && sick1.isEnd() && doc1.isEnd())
        std::cout << "Lists cleaned up." << std::endl;
    else
        std::cerr << "You fail ! Boo !" << std::endl;
    return (0);
}
```

Terminal

```
~/B-CPP-300> ./a.out

Dr.Cox: I'm Dr.Cox! How do you kreog?
Dr.House: I'm Dr.House! How do you kreog?
Dr.Boudur-Oulot: I'm Dr.Boudur-Oulot! How do you kreog?
[HOSPITAL] Doctor Cox just arrived!
Dr.Cox: Time to get to work!
[HOSPITAL] Doctor House just arrived!
Dr.House: Time to get to work!
[HOSPITAL] Doctor Boudur-Oulot just arrived!
Dr.Boudur-Oulot: Time to get to work!
[HOSPITAL] Patient Ganepar just arrived!
[HOSPITAL] Patient Varia just arrived!
[HOSPITAL] Patient RedFace just arrived!
[HOSPITAL] Patient Scarface just arrived!
[HOSPITAL] Patient Falter just arrived!
[HOSPITAL] Nurse 1 just arrived!
Nurse 1: Time to get to work!
[HOSPITAL] Nurse 2 just arrived!
Nurse 2: Time to get to work!
[HOSPITAL] Work starting with:
Doctors: Cox, House, Boudur-Oulot.
Nurses: 1, 2.
Patients: Ganepar, Varia, RedFace, Scarface, Falter.

Dr.Cox: So what's goerking you Mr.Ganepar?
Mr.Ganepar: Gooeeeeerrk!!
```


Terminal
Nurse 1: Kreog! Mr.Ganepar needs a Kinder!
Mr.Ganepar: There is a toy inside!
Dr.House: So what's goerking you Mr.Varia?
Mr.Varia: Gooeeeeeerrk!!
Nurse 2: Kreog! Mr.Varia needs a Mars!
Mr.Varia: Mars, and it kreogs!
Dr.Boudur-Oulot: So what's goerking you Mr.RedFace?
Mr.RedFace: Gooeeeeeerrk!!
Nurse 1: Kreog! Mr.RedFace needs a Kinder!
Mr.RedFace: There is a toy inside!
Dr.Cox: So what's goerking you Mr.Scarface?
Mr.Scarface: Gooeeeeeerrk!!
Nurse 2: Kreog! Mr.Scarface needs a Kinder!
Mr.Scarface: There is a toy inside!
Dr.House: So what's goerking you Mr.Falter?
Mr.Falter: Gooeeeeeerrk!!
Nurse 1: Kreog! Mr.Falter needs a Crunch!
Mr.Falter: Goerkreog!
Nurse 1: Time to go home to my eucalyptus forest!
Nurse 2: Time to go home to my eucalyptus forest!
Dr.Cox: Time to go home to my eucalyptus forest!
Dr.House: Time to go home to my eucalyptus forest!
Dr.Boudur-Oulot: Time to go home to my eucalyptus forest!
Lists cleaned up.
Mr.Falter: Kreoooogg!! I'm cuuuured!
Mr.Varia: Kreoooogg!! I'm cuuuured!
Mr.RedFace: Kreoooogg!! I'm cuuuured!
Mr.Scarface: Kreoooogg!! I'm cuuuured!
Mr.Ganepar: Kreoooogg!! I'm cuuuured!
Nurse 2: Finally some rest!
Nurse 1: Finally some rest!