Hospital

Goal: I will learn modularity in object-oriented programming and how classes (objects) can be related to each other. I will practice reading existing code and to use pointers and dynamic memory management. At the same time, I will revisit STL containers. I will also consider how to divide program into parts by using functions and classes.

Instructions: Retrieve the code template: templates/11/hospital/ -> student/11/hospital/. The code template contains rather many files and classes, but they are very simple and easy to be understood.

Some of the commands have already been implemented in the template code. You can start by testing their functionality.

Important

Before starting to write code, read carefully the whole assignment. Especially, note the sections *Implementing the program in stages* (required commits) and *Special requirements*.

Note

This project will be done independently. Working in pairs is not allowed, instead it will be considered as plagiarism.

Complete the program that reads user-given commands. Depending on the command, the program either stores data into a suitable data structure, remove data from it, or makes searches in the data structure in question.

Header comment and feedback language

In the same way as in the first project, the header comment is again required. This time we provide no ready-made header comment, but you should write it by yourself. You can use the first project's assignment as an example (see 4.6 (P) Pairs).

The feedback language will be chosen in the submission box (at the very end of this page). **By default, the feedback language is Finnish**, but you can change it, if you want to have the feedback, given by an assistant, in English. We will use the language you have given in the submission box of the final submission.

Program description

The program consists of classes Hospital, CarePeriod, Person, Date, and Cli, as well as the module Utils.

The objects (instances) of the class Person can be either patients or hospital staff (but not both). Staff members are not separated more precisely, e.g. for doctors and nurses. Persons can be identified by their names (as has been done in example executions below), or any string.

Hospital staff is permanent. After recruitment, a staff member stays until the end of the program. (The program has a command for recruiting but there is no command for removing a staff member.) However, a patient can be added and removed. When a patient is added, a new care period will be created, too, and all care periods created stay until the end of the program. A patient can have several care periods during the program. In other words, a patient may enter and leave hospital many times during the program.

The class CarePeriod describes a single care period of a single patient, but as told above, one patient may have several care periods. A care period has at least a start date, and closed care periods have also an end date. Besides a patient, a care period has 0-n staff members assigned to. A person can act as a staff member in several care periods, each of which can be that of a different or the same patient.

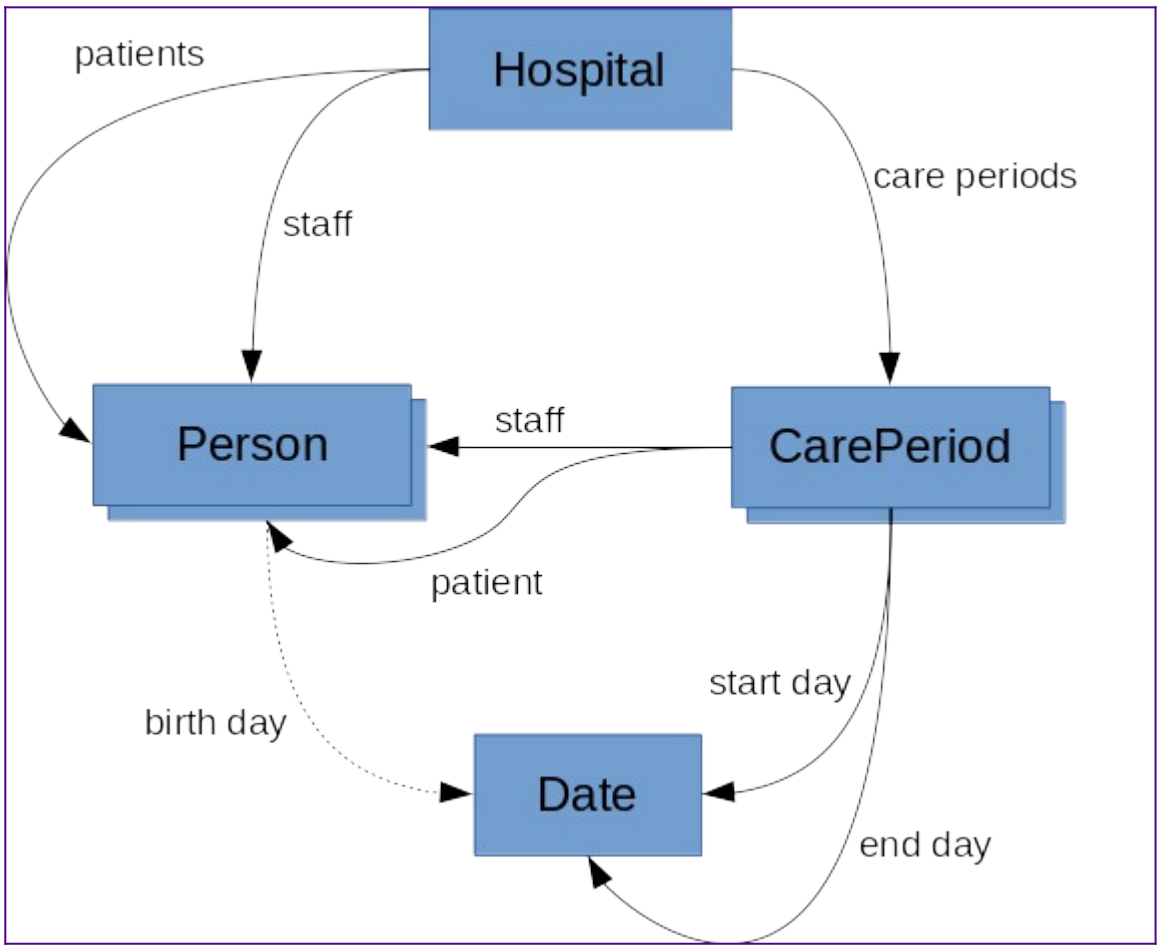
A patient may have medicines. In the program, a medicine is described as a struct, not

as a class, since a medicine has no operations. This struct can be found from class Person. Medicines are specific for a patient, and they stay between care periods. When (and only when) a patient is having a care period in the hospital, new medicines can be prescribed for them, or medicines can be removed from them, but after a care period a patient has those medicines that have been prescribed for them and not removed. Instead, the staff members of a patient are specific for care period, and in different care periods the patient may have different (or same) staff assigned to.

The class Cli (command line interpreter) manages the commands used in the program. The module Utils provides general functions. No changes are needed in either of these modules (Cli or Utils). They, for example, check if the user has given a correct amount and type of parameters for each command. These checks are already implemented, so you need not care about them.

The class Hospital manages the program in whole, and only one instance is created from it. A hospital has patients, staff, and care periods.

The relationships between classes can be seen in the figure below.



There is an arrow from class A to class B, if A knows B. This means that the instance of B (or a pointer to it) is an attribute of A. The class Cli and the module Utils have been left away from the figure, since they are utility modules, only weakly belonging to the hospital system. Over the arrows you can see descriptive texts to make it easier to understand, why the relation in question is needed.

In the figure, the birth day arrow is drawn as a dashed line, since such a relation would be very natural, but the program does not really use it.

As said above, the same person can occur in several places. For example, the same patient can act as a patient in several care periods, and the same they can act as a staff member in several care periods. Therefore, you need pointers. Each person is created only once, and when you need this person in several actions, you will use a pointer to the person.

General information about the commands

The program has many commands, but your task is to implement only about half of them.

No input file is given for the program, and thus, the hospital is empty at start, i.e. there are no staff nor patients. All insertions (as well as removals) are done with commands. However, among the commands, there is READ_FROM for reading commands from a file (see command 3 in commands already implemented).

At start and each time the user is expected to give input, the program prints the prompt:

```
Hosp>
```

To this prompt, the user can give commands introcuded below (or just press Enter). The commands can be written by using lower-case letters, upper-case letters, or by mixing them. Therefore exit command can be written, for example, as QUIT, quit, or Quit. Each command has an acronym consisting of a couple of letters. You can find the acronyms at the description of the HELP command below. For example, the acronym of QUIT is Q (or q as a lower-case letter).

Each command has a predefined number of parameters. You can see the parameters at each command description enclosed by angle brackets. If a parameter consists of several words, it must be enclosed by quote marks. If too less or too much parameters are given, the program prints the error message:

```
Error: Wrong amount of parameters.
```

Checking the amount of parameters is implemented in the template code, and thus, you need not care about it.

If the user gives other command than that known by the program, the following error message is given:

```
Hosp> something

Error: Unknown commands given.
```

```
Hosp>

Hosp>
```

After completing the execution of a command given by the user, the program prints the prompt again, until the user gives the command QUIT.

If the user just prints Enter without giving any command, the prompt is printed again, as shown in the latest example. In this sense, the program works like the command line in Linux desktop.

The next two sections first describe implemented commands and after that commands to be implemented. Naturally, there is no need to read the first one of these sections carefully, since you only need to know how to use the commands. However, the latter one of the sections must be read carefully, since it is your task to implement the commands.

The commands already implemented with their error messages

1. QUIT - The program terminates with the return value EXIT_SUCCESS without printing anything. The command takes no parameters, but if such are given, they will be ignored.

2. HELP - The command prints all available commands with their acronyms. Each line first shows a command description and then a colon followed by the command and its acronym:

```
Hosp> HELP

Recruit staff : RECRUIT R

Take patient to hospital : ENTER E

Take patient from hospital : LEAVE L

Assign staff for a patient : ASSIGN_STAFF AS

Add medicine for a patient : ADD_MEDICINE AM

Remove medicine from a patient : REMOVE_MEDICINE RM
```

```
Print patient's info : PRINT_PATIENT_INFO PPI

Print care periods per staff : PRINT_CARE_PERIODS PCPS

Print all used medicines : PRINT_ALL_MEDICINES PAM

Print all staff : PRINT_ALL_STAFF PAS

Print all patients : PRINT_ALL_PATIENTS PAP

Print current patients : PRINT_CURRENT_PATIENTS PCP

Set date : SET_DATE SD

Advance date : ADVANCE_DATE AD

Read : READ_FROM RF

Help : HELP H

Quit : QUIT Q

Hosp>
```

3. READ_FROM <file> - The command reads other commands with their parameters from the given file, which makes testing the program easier. The template code includes the file assignment.input, which includes some of the commands presented in the example executions. The file attached uses acronyms of the commands. The content of the file is as follows:

```
e Pekka

r Jussi

ah Jussi Pekka

ad 2

l Pekka

e Pekka

am Burana 200 2 Pekka

q
```

An input file can be given as a parameter for the reading command as follows:

```
Hosp> READ_FROM assignment.input

Input read from file: assignment.input

Hosp>
```

In this way, you can execute all successful commands in the file faster than writing them one by one after the prompt.

Note that the file assignment.input includes also such commands that have not been implemented in the template code.

If the input file includes an erronous command, e.g. a command with a wrong number of parameters, no error message is given, as will be given in the command line. Moreover, the last command in the file must be QUIT (or its acronym), since otherwise reading the file never stops. Therefore it is important to follow these instructions, if you use an input file written by yourself when testing the program. **Testing the program (in one way or another) is highly recommended.**

If the file given as a parameter is unknown, the program prints the error message:

```
Hosp> READ_FROM not_found.txt

Error: Can't read given file.
```

After this the program terminates its execution with the return value EXIT_SUCCESS.

4. SET_DATE <dd> <mm> <yyyy> - The command sets the date to be the date dd.mm.yyyy. (The current date is set as 24.2.2021 in the file utils.hh, and if you wish, you can change it also in the code, but the current command makes the same thing.) The command takes three integer parameters. If the number of given day (or month) is 1, it can be given either as 1 or 01, and the same holds for other 1-digit numbers. For example:

```
Hosp> SET_DATE 1 04 2021
```

```
Date has been set to 1.4.2021

Hosp>
```

If the number given as a parameter is too great to be able to be a correct day or month, the number in question will be 1. For example:

```
Hosp> SET_DATE 32 13 2021

Date has been set to 1.1.2021

Hosp>
```

If the given parameters are something else than positive integer numbers, the program prints the error message:

```
Hosp> SET_DATE 1 3 w

Error: Wrong type of parameters.
```

5. ADVANCE_DATE <non-negative integer> - The command advances the current date with the given number of days ahead. For example:

```
Hosp> ADVANCE_DATE 7

New date is 3.3.2021

Hosp>
```

By using zero as the parameter, the command reveals the current date. For example:

```
Hosp> ADVANCE_DATE 0

New date is 24.2.2021

Hosp>
```

If the given parameter is something else than a non-negative integer number, the program prints the error message:

```
Hosp> ADVANCE_DATE -1
```

Error: Wrong type of parameters.

Hosp>

6. RECRUIT <id> - The command adds a new staff member to the hospital. The command requires one parameter that can be a name. (The given id will be used as a parameter for other commands.) An example on recruiting staff:

Hosp> RECRUIT Jussi

A new staff member has been recruited.

Hosp>

If a staff member with the given id already exists, the program prints the error message:

Hosp> RECRUIT Jussi

A new staff member has been recruited.

Hosp> RECRUIT Jussi

Error: Already exists: Jussi

Hosp>

7. PRINT_ALL_STAFF - The command prints all staff recruited, members are listed in alphabetical order, one below another. If there is no staff in the hospital, the program prints None. For example:

Hos> PRINT_ALL_STAFF

None

Hosp> RECRUIT Jussi

A new staff member has been recruited.

Hosp> PRINT_ALL_STAFF

Jussi

Hosp> RECRUIT Jukka

A new staff member has been recruited.

```
Hosp>  PRINT_ALL_STAFF
Jukka
Jussi
```

8. ADD_MEDICINE <medicine name> <strength> <dosage> <patient id> - The command adds a medicine with a given strength (as milligrams) and dosage for a patient. The strength and dosage must integer numbers.

   Medicines can be prescribed only for those patients that are currently in hospital. The following examples assumes that Pekka is a patient in the hospital:

```
Hosp> ADD_MEDICINE Burana 200 2 Pekka
Medicine added for: Pekka
Hosp>
```

   If the given medicine has already been prescibed for the patient, the command can be used to change the prescription (strenth and/or dosage). If the whole prescription (name, strenth, dosage) is the same as the patient already has, nothing happens and no error message is given. For example:

```
Hosp> ADD_MEDICINE Burana 400 2 Pekka
Medicine added for: Pekka
Hosp> ADD_MEDICINE Burana 400 2 Pekka
Medicine added for: Pekka
Hosp>
```

   If the strength or dosage is something else than integer numbers, the program prints the error message:

```
Hosp> ADD_MEDICINE Voltaren xx yy Pekka
Error: Wrong type of parameters.
Hosp>
```

If the given patient cannot be found (they are not in the hospital at the moment), the program prints the error message:

```
Hosp> ADD_MEDICINE Voltaren 100 1 someone
Error: Can't find anything matching: someone
Hosp>
```

9. REMOVE_MEDICINE <medicine name> <patient id> - The command removes a medicine from a patient. A medicine can be removed only from a patient that is currently in hospital. The following examples assumes that Pekka is a patient in the hospital:

```
Hosp> REMOVE_MEDICINE Burana Pekka
Medicine removed from: Pekka
Hosp>
```

If the given medicine has not been described for the patient, nothing happens and no error message is given. For example:

```
Hosp> REMOVE_MEDICINE Burana Pekka
Medicine removed from: Pekka
Hosp> REMOVE_MEDICINE Burana Pekka
Medicine removed from: Pekka
Hosp>
```

If the given patient cannot be found (they are not in the hospital at the moment), the program prints the error message:

```
Hosp> REMOVE_MEDICINE Burana someone
Error: Can't find anything matching: someone
Hosp>
```

Commands to be implemented with their error messages

1. ENTER <id> - The command adds a new patient in the hospital. The command requires one parameter that can be the name of the patient. (The given id will be used as a parameter for other commands.) An example on adding a patient:

```
Hosp> ENTER Pekka

A new patient has entered.

Hosp>
```

   When a patient enters hospital, a new care period is created such that its start date will be current value of today variable of Utils module.

   If a patient with the given id already exists, the program prints the error message:

```
Hosp> ENTER Pekka

A new patient has entered.

Hosp> ENTER Pekka

Error: Already exists: Pekka

Hosp>
```

   However, if a patient leaves the hospital (with the command LEAVE), they enter again (to a new care period). In such a case, a new patient is not created, but the program stores data about all patients visited the hospital during the same execution. The data is not stored between executions, but when starting a re-run, there are at first no staff nor patients in the hospital.

   A new care period is always created, regardless of the patient has earlier visited the hospital or not.

2. LEAVE <id> - The command removes a patient from the hospital, but the data about their care period is kept (dates and staff of the care period). For example:

```
Hosp> LEAVE Pekka
```

```
Patient left hospital, care period closed.

Hosp>
```

When a patiet leaves the hospital, their current care period is closed, whereupon an end date can be set for the care period. The end date is the current date.

If the patient to be removed cannot be found (they are not in the hospital at the moment), the program prints the error message:

```
Hosp> LEAVE Pekka

Patient left hospital, care period closed.

Hosp> LEAVE Pekka

Error: Can't find anything matching: Pekka

Hosp>
```

The following example shows how a patient can first enter hospital, then leave it, and enter again:

```
Hosp> ENTER Pekka

A new patient has entered.

Hosp> ENTER Pekka

Error: Already exists: Pekka

Hosp> LEAVE Pekka

Patient left hospital, care period closed.

Hosp> ENTER Pekka

A new patient has entered.

Hosp>
```

3. ASSIGN_STAFF <staff member id> <patient id> - The command assigns the given staff member to work in the given patient’s current care period. For example:

```
Hosp> ENTER Pekka
A new patient has entered.
Hosp> RECRUIT Jussi
A new staff member has been recruited.
Hosp> ASSIGN_STAFF Jussi Pekka
Staff assigned for: Pekka
Hosp>
```

If the same staff member is tried to be assigned to the same patient again, nothing happens, and no error message is given:

```
Hosp> ASSIGN_STAFF Jussi Pekka
Staff assigned for: Pekka
Hosp> ADD_STAFF Jussi Pekka
Staff assigned for: Pekka
```

The following example assumes that the commands above have been executed first. If staff member or patient cannot be found, the program prints the error message shown below:

```
Hosp> ASSIGN_STAFF Jukka Pekka
Error: Can't find anything matching: Jukka
Hosp>
Hosp> LEAVE Pekka
Patient left hospital, care period closed.
Hosp> ASSIGN_STAFF Jussi Pekka
Error: Can't find anything matching: Pekka
Hosp>
Hosp> ASSIGN_STAFF Jukka Pekka
```

```
Error: Can't find anything matching: Jukka

Hosp>
```

More generally, if the first id cannot be found, the program informs about it. If the first id is found but not the second one, the program informs about the second one. If neither of the ids cannot be found, the program informs only about the first one.

4. PRINT_PATIENT_INFO <patient id> - The command prints information about patient’s all care periods and their medicines. From each care period, staff and start date are printed, and from closed care periods also the end date is printed. Care periods are printed in chronological order (the earlier one first). Staff members and medicines are printed in alphabetical order. If the patient has no staff members assigned or no medicines, the word None is printed instead. For example:

```
Hosp> ENTER Pekka

A new patient has entered.

Hosp> PRINT_PATIENT_INFO Pekka

* Care period: 24.2.2021 -

  - Staff: None

* Medicines: None

Hosp>

Hosp> ADD_MEDICINE Burana 200 2 Pekka

Medicine added for: Pekka

Hosp> ADD_MEDICINE Panadol 500 1 Pekka

Medicine added for: Pekka

Hosp>

Hosp> RECRUIT Jussi

A new staff member has been recruited.
```

Hosp> RECRUIT Jukka

A new staff member has been recruited.

Hosp>

Hosp> ASSIGN_STAFF Jussi Pekka

Staff assigned for: Pekka

Hosp> ASSIGN_STAFF Jukka Pekka

Staff assigned for: Pekka

Hosp>

Hosp> PRINT_PATIENT_INFO Pekka

* Care period: 24.2.2021 -

  - Staff: Jukka Jussi

* Medicines:

  - Burana 200 mg x 2

  - Panadol 500 mg x 1

Hosp>

Hosp> ADVANCE_DATE 2

New date is 26.2.2021

Hosp> LEAVE Pekka

Patient left hospital, care period closed.

Hosp> ADVANCE_DATE 3

New date is 1.3.2021

Hosp>

Hosp> ENTER Pekka

A new patient has entered.

Hosp> PRINT_PATIENT_INFO Pekka

* Care period: 24.2.2021 - 26.2.2021

```
    - Staff: Jukka Jussi
  * Care period: 1.3.2021 -
    - Staff: None
  * Medicines:
    - Burana 200 mg x 2
    - Panadol 500 mg x 1
Hosp>
```

If the given patient cannot be found, the program prints the error message:

```
Hosp> PRINT_PATIENT_INFO someone
Error: Can't find anything matching: someone
Hosp>
```

The command works for all patient visited the hospital at some time. In other words, a patient cannot be found only if they have never visited the hospital.

As told in the program description, medicines are specific for patients, and thus, they do not depend on care periods. Instead, staff assigned to the patient is specific for care periods.

5. PRINT_CARE_PERIODS <staff member id> - The command prints those care periods where the given staff member has worked or works. From each care period, the patient and start date are printed, and from closed care periods also the end date is printed. Care periods are printed in chronological order (the earlier one first). If the given staff member has been recruited but not assigned to work in any patient's care period (with the command ASSIGN_STAFF), the program prints None. For example:

```
Hosp> RECRUIT Jussi
A new staff member has been recruited.
Hosp> PRINT_CARE_PERIODS Jussi
```

```
None

Hosp> ENTER Pekka

A new patient has entered.

Hosp> ENTER Matti

A new patient has entered.

Hosp> ASSIGN_STAFF Jussi Pekka

Staff assigned for: Pekka

Hosp> ASSIGN_STAFF Jussi Matti

Staff assigned for: Matti

Hosp> ADVANCE_DATE 2

New date is 26.2.2021

Hosp> LEAVE Matti

Patient left hospital, care period closed.

Hosp>

Hosp> PRINT_CARE_PERIODS Jussi

24.2.2021 -

* Patient: Pekka

24.2.2021 - 26.2.2021

* Patient: Matti

Hosp>
```

If the given staff member cannot be found, the program prints the error message:

```
Hosp> PRINT_CARE_PERIODS someone

Error: Can't find anything matching: someone

Hosp>
```

6. PRINT_ALL_MEDICINES - The command prints all medicines currently used by some patient that has visited hospital at some time or that lies there at the moment. Medicines are listed in alphabetical order, and with each medicine also its user is told. Patients using the same medicine are listed in alphabetical order. Medicines are the same, if they have the same name, strength and dosage do not matter. If no medicine is in use, the word None is printed. For example:

```
Hosp> PRINT_ALL_MEDICINES
None
Hosp> ENTER Pekka
Hosp> ADD_MEDICINE Burana 200 2 Pekka
Medicine added for: Pekka
Hosp> ADD_MEDICINE Panadol 500 1 Pekka
Medicine added for: Pekka
Hosp>
Hosp> ENTER Matti
Hosp> ADD_MEDICINE Burana 400 2 Matti
Medicine added for: Matti
Hosp>
Hosp> PRINT_ALL_MEDICINES
Burana prescribed for
* Matti
* Pekka
Panadol prescribed for
* Pekka
Hosp>
Hosp> REMOVE_MEDICINE Burana Pekka
Medicine removed from: Pekka
```

```
Hosp> PRINT_ALL_MEDICINES

Burana prescribed for

* Matti

Panadol prescribed for

* Pekka

Hosp>

Hosp> LEAVE Pekka

Patient left hospital, care period closed.

Hosp> PRINT_ALL_MEDICINES

Burana prescribed for

* Matti

Panadol prescribed for

* Pekka

Hosp>
```

7. PRINT_ALL_PATIENTS - The command prints information about all patients visited the hospital at some time, the current patients are also included. Information about the patients are listed in alphabetical order based on the patient id.

   The command gives the same output as the command PRINT_PATIENT_INFO, but like this command was executed as many times as there have been patients in the hospital. Before printing infomation about a patient, the patient’s id is printed first.

   Assuming that the actions with the previous command (PRINT_ALL_MEDICINES) have been executed, the program prints:

```
Hosp> PRINT_ALL_PATIENTS

Matti
```

```
* Care period: 24.02.2021 -
   - Staff: None
* Medicines:
   - Burana 400 mg x 2
Pekka
* Care period: 24.02.2021 - 24.02.2021
   - Staff: None
* Medicines:
   - Panadol 500 mg x 1
Hosp>
```

If no patient had ever visited the hospital, the output of the command would look like as:

```
Hosp> PRINT_ALL_PATIENTS
None
Hosp>
```

8. PRINT_CURRENT_PATIENTS - The command prints information about all current patients in the hospital. Information about the patients are listed in the alphabetical order based on the patient id.

   The command gives the same output as the command PRINT_PATIENT_INFO, but like this command was executed as many times as there are patients in the hospital at the moment. Before printing infomation about a patient, the patient’s id is printed first.

   Assuming that the actions with the earlier command (PRINT_ALL_MEDICINES) have been executed, the program prints:

```
Hosp> PRINT_CURRENT_PATIENTS
Matti
```

```
  * Care period: 24.02.2021 -
    - Staff: None
  * Medicines:
    - Burana 400 mg x 2
Hosp>
```

If there were no patients at the moment in the hospital, the output of the command would look like as:

```
Hosp> PRINT_CURRENT_PATIENTS
None
Hosp>
```

When quitting the program, it prints the persons, whose memory cells are deallocated. This can be useful when you test your program with your own test cases. If the program had persons appeared in the earlier examples, it prints:

```
Hosp> QUIT
Person Jukka destructed.
Person Jussi destructed.
Person Matti destructed.
Person Pekka destructed.
Press <RETURN> to close this window...
```

The automatic tests assume that there are no such prints, and thus, before Plussa submission you should remove the only code line in the destructor of Person class.

Modules of the program

The code template consists of the main program module and six other modules. The main program module is very simple. It just starts the command line interpreter (CLI) that goes on until the QUIT command is given by the user. There is no need to change

the main program.

The program contains the module Utils that is not a class but a namespace. It provides the utility function split that you should be familiar with, the function is_numeric, and the date today. If you wish, you can update the date value in the file utils.hh. Currently its value is 24.2.2021. Otherwise, there is no need to change this module. The date can also be changed with the command SET_DATE.

The command line interpreter, i.e. the class Cli has been defined and implemented in files cli.hh and cli.cpp. This class is completely implemented in the code template, and you need not modify it. The purpose of the class is to identify the commands from the user input. All the commands will be implemented in the Hospital class. The header file of the command line interpreter defines a command vector including function pointers. From it you can see which function in class Hospital implements each command. The other parts of the functionality of command line interpreter are not necessary to understand. Especially, knowing/understanding function pointers is not required.

The class Date describes dates. Each part of a date (day, month, year) is presented as an integer. A day can be moved forward by calling the method advance, which advances the date by the given number of days. The method is_default tells if the date is a default one, i.e. a date with zero as the value of day, month, and year. If a care period has not yet closed, its end date is the default one. Therefore, you can use is_default method to check, if a care period has ended or not. Moreover, Date class provides methods for setting a date and for printing it. It is possible to compare the equality between dates. In addition, the class provides the < operator: date a is less than date b, if a precedes b. (The current code does not use comparison operator functions.) You need not change the class Date.

The class Person describes persons: patients and hospital staff. Persons can be identified based on attribute id_. The examples given earlier used names as identifiers, which requires unique names of persons. Basically, a patient and a staff member could have the same name, but then they would be considered different persons (different objects). The class Person has the attribute date_of_birth_, but it is not actually necessary. The attribute medicines_ is a map containing medicines prescribed for the person (patient). Medicines cannot be prescribed for staff, since they can be

prescribed only for patients that are currently in hospital. The class Person need not be changed, unless you want to add there new methods, which you call from other methods you have implemented in other classes. (However, remember to remove the print line from the destructor of the class, as adviced at the end of the previous section.)

The program has the class CarePeriod for describing care periods. It is your task to implement the class.

The class Hospital describes a hospital (in general, there can be several of them, but in this program, we need only one). The class contains information about all the care periods and persons in the hospital. As said earlier, a person can be a patient or a staff member. For storing this data, the class has containers that are added with new elements based on user-given commands. All the commands targeting to the hospital will be implemented in this class.

Note that a patient may have several care periods, and a staff member may work in several care periods. In such cases, only one Person object is created, and the object is pointed from several places (data structures). In such situations, pointers are essential.

The relationships between modules (classes) have been described in the figure at the beginning of the assignment.

The assignment in more detail

Your task is to complete the classes Hospital and CarePeriod such that commands work in the way described above. You can change also the other classes if needed. You can, for example, feel it necessary to add new methods in a class. Most of the data structures are given in the code template.

The class CarePeriod is almost empty, and thus, you can design and implement it totally by yourself. The class Hospital has no implementations for the methods:

- enter
- leave

- assign_staff
- print_patient_info
- print_care_periods
- print_all_medicines
- print_all_patients
- print_current_patients.

The purpose of each above is to implement a command with the same name.

Feel free to implement other utility functions, as well as add new attributes.

Program code (e.g. printing code) should be placed in the class, the data of which is processed (printed). For example, a printing chain can start from a ”bigger” class, which prints something by itself and then asks its parts to print their portion. Here a bigger class means a class, which has intances of other classes as its parts (attributes).

While designing and implementing the assignment, it is especially important to pay attention to how instances (objects or pointers to them) of one class can be attributes of another class.

The program has different kinds of persons: staff, patients currently in hospital, and patients earlier visited hospital. All of them are instances of the Person class. You can distinguish different persons by adding new attributes in the Person class. Their types can, for example, be bool or enum. Another (and perhaps better) option is to collect different persons in different containers. For example, there is the attribute current_patients_ in the class Hospital that contains those patients that are in the hospital at the moment.

Command functions (a method that implements a command) have the parameter params, the type of which is Params. This means a vector, the definition of which is in the file hospital.hh. The number of parameters are checked in the method exec of

class Cli. Therefore you can assume that the vector params has exactly the same number of elements as the command in question requires.

However, some commands have no parameters, whereupon the corresponding command functions do not use the parameter params they receive. In such cases, the name of the parameter is left away, but not the type of it. The type cannot be left away, since these functions are called via a function pointer. All functions called via a function pointer of the same type must have the same number of parameters of the same type. All these things have been implemented in the template code, and thus, you need not care about this. **Implementing the program does not require knowing anything about function pointers.**

Tips

- Start by studying the ready-made code. You need not understand all the details in the given code. It does not matter, if you do not understand code of Cli. From the other classes, you should find out, which methods they provide (public methods) and which attributes they have. Explore the implementation details of methods only if/when you notice the need for these details. In most cases it is enough that you conclude the purpose of a method from its name.
- Before implementing the program, think carefully how to structure the program to be able fulfill all the requirements.
- The main things to be practiced in this project are modularity and pointers. However, you need add a couple of STL containers but choosing them should not be a big issue at this point.
    - What would be a suitable container for the care periods of a hospital, if the aim is to go through the care periods in the order, in which they have been added?
    - What would be a suitable container for the staff of a care period, if the aim is to print the staff members in alphabetical order?
- Note that const functions inside a class cannot call those functions of the same class that are not defined as const. You can notice this by the compilation error [-fpermissive]. In other words, a function that cannot change the state of the object, cannot call a function that is able to change the state of the object.

- Recall that as a default, the program will be compiled into a build-directory. So, for testing purposes, you can move possible input data files to this same directory.

Implementing the program in parts

In this phase of the course, you should have a conception on, in which parts to implement the program and which commits to use.

Special requirements

If you want your assignment to pass the evaluation, you must meet these requirements:

- You must use dynamic memory management in your program. However, it must not have errors related to memory management. Therefore executing valgrind is recommended. Pointers used in the program can be normal pointers, smart pointer, or both of them.
- Containers only from STL library are allowed, not e.g. from Boost.

Evaluation

To end up to assistants' evaluation, your work must first pass the automatic tests. If the automatic tests give 0 points for your work, also your final points will be 0, and your work will not be evaluated by an assistant.

The assistant evaluates the submissions that have passed the automated testing (= 1 p) **and fulfilled the special requirements of the assignment** based on the last commit before the deadline, according to the following criteria:

- The overall principle of the solution: 0-40 points:
    - The learning goals of the exercise have been achieved.
    - The program code has been split into logical, suitably long segments using functions, classes, and/or methods.
    - If the program uses classes and objects, these have been implemented

according to the basics of object-oriented programming (see from round 4 the section ”About programming style” at ”Object-oriented programming”).

    - The data structure does not include repetitive data nor unnecessary parts. The chosen data structures have been used in a reasonable way.
    - The program code does not include unnecessary repetitions nor other unnecessary parts.
    - The program code does not include unnecessary limitations or assumptions or other forced solutions.
    - The implementations of the program’s structures are easy to understand.
    - Global variables have not been used in the program code (global constants are OK).
    - The program does not terminate with the exit function.
    - The program has no errors related to memory management (use valgrind and note the updates in section 10.1).

- Programming style: -20-0 points:
    - Variables and functions are named clearly and appropriately.
    - Named constants have been used instead of magic numbers.
    - The program code is neatly formatted.
    - Each program line has at most 80 characters.
    - At the beginning of each file written/edited by yourself, there is a comment explaining the purpose of the file, your name and student number and other necessary information (see the assignment of the first project on round 4).
    - At the beginning of each function/method (in the header file if possible), there is a comment describing its working, return value, and parameters.
    - There are comments in the code where necessary.

        - All the variables have been initialized.
        - The compiler does not give warnings while compiling.
- Using the version control: 0-10 points:
        - There are enough commits.
        - The content of commit messages is clear and relevant.

From the above list, please note that in this phase of the course it is assumed that the student follows good programming style. Therefore, you will not get points from following the style guidelines, but if you do not follow them, your points will be decreased. So, it is recommended to revisit the material about programming style on rounds 4 and 11.