Systems Programming – Assignment #1
BGU fitness studio management system

2022/Fall

---

## 1 Before You Start

- It is mandatory to submit all the assignments in pairs. It is recommended to find a partner as soon as possible.
- Submission is made via moodle in a single zip file called ”student1ID-student2ID.zip” - Only one partner should submit via moodle (no need for duplicate submissions)
- Read the assignment together with your partner and make sure you understand the tasks. Please do not ask questions before you have read the whole assignment.
- Skeleton classes will be provided on the assignment page. You must use these classes as a basis for your project and implement (at least) all the declared functions.

KEEP IN MIND

While you are free to develop your project in whatever environment you want, your project will be tested and graded ONLY on a CS LAB UNIX machine. It is your responsibility to deliver a code that compiles, links and runs on it. Failure to do so will result in a grade of 0 on your assignment.

Therefore, it is mandatory that you compile, link, and run your assignment on a lab UNIX machine before submitting it.

We will reject, upfront, any appeal regarding this matter!!

We do not care if it runs on any other Unix/Windows machine.

Please remember that it is unpleasant for us to fail your assignments, at least as it is for you. Just do it the right way.

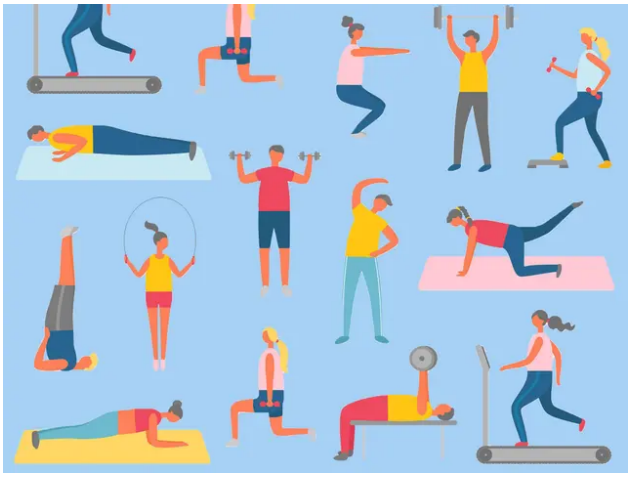
## 2 Assignment Goals

The objective of this assignment is to design an object-oriented system and gain implementation experience in C++ while using classes, standard data structures, and unique C++ properties such as the “Rule of 5”. You will learn how to handle memory in C++ and avoid memory leaks. The resulting program must be as efficient as possible.

## 3 Assignment Definition

In our studio, we offer many workout plans (options). We have talented trainers who are experts in all of our workout options (Yoga, Pilates, CrossFit, etc.), where each exercise can be Cardio Anaerobic or Mixed. For example, Yoga is considered an anaerobic exercise, while spinning (cycling) is a mixed exercise.

In this assignment, you will write a C++ program that simulates a fitness studio management system. The program will open the studio, assign customers to trainers, make a workout plan that consists of one or more workout options, calculate the trainer's salary, and other requests as described below.
The program will get a config file as an input, including all required information about the fitness studio opening - the number of trainers, how many customers each trainer can handle, and details about the studio's possible workouts. A detailed description of the input file will be defined in section 3.5.



Each trainer is capable of performing all of the suggested workout types. There are four types of customers in this studio. Each customer type has a unique workout strategy. A customer may request a workout from a trainer more than once. In such cases, some customers may order an additional and (maybe) different activity. In part 3.3, there is a detailed description of those strategies.

Each trainer in the studio has a limited amount of available spots in their workout session (provided in the config file). Each trainer can handle multiple customers with more than one type of workout simultaneously. In this studio, it's **impossible** to add new customers to an open workout session (open trainer), but it's **possible** to move customers from one workout session to another between different trainers under the trainer’s capacities.

The trainer's salary is the total price of all workout that was ordered for that trainer.

### 3.1 The Program flow

The program receives the path of the config file as the first command-line argument

```
string configurationFile = argv[1]
```

The config file contains the info about the number of trainers, the trainer’s capacities, and the workout that is available in the studio (workout options). Further described in section 3.5.

Once the program starts, it opens the studio by calling the `start()` function, followed by printing `"Studio is now open!"` to the screen.

Then the program waits for the user to enter an action to execute. After each executed action, the program waits for the next action in a loop. The program ends when the user enters the action `"closeall"` (See actions in part 3.4).

## 3.2 Classes

**Studio** – This class holds a list (Vector) of trainers and workout plans that the studio offers.

**Trainer** – This class represents a trainer in the studio. Each trainer has a finite number of available spots (int capacity, provided in the config file). It also holds a status flag `bool open` that indicates whether the trainer's workout session is in progress, a list of workout plans done for this trainer's workout session, and a list of customers. Trainer id's start from 0.

**Workout** – This class represents a workout from the workout plan. It has an id, name, price, and type. The ID of a workout is the order in which he appeared in the config file (will be further described).

- `enum WorkoutType` - An enumeration is a user-defined data type that consists of integral constants. Define an enumeration by using the keyword enum. By default, the first element is 0, and the second is 1, and so on. You can change the default value of an enum element during declaration (if necessary).

**Customer** – This is an abstract class for the different customer classes. There are several types of customers, and each of them has a different workout ordering strategy. Each customer who arrives at the studio will get a number (id) that will serve as an identifier **as long as** he participates in a workout session. This number will be a serial number of all customers that arrived so far, starting from 0 (the first customer will get 0, the second customer will get 1, etc.). Note that this "id" serves as a temporary identifier- if a customer leaves a workout session and then comes back, he will get new ID (ID will appear only once).

The class has a pure virtual method `order(workout_options)`, which receives the workout_options and returns a vector of workouts that the customers ordered. Ordering strategies are further described in 3.3.

**BaseAction** – This is an abstract class for the different action classes. The motivation for keeping a BaseAction class is to enable logging multiple action types. The class contains a pure virtual method `act(Studio& studio)` which receives a reference to the studio as a parameter and performs an action on it; A pure virtual method `toString()` which returns a string representation of the action; A flag which stores the current status of the action: "**Completed**" for successfully completed actions, and "**Error**" for actions that couldn't be completed.
After each action is completed- if the action was completed successfully, the protected method `complete()` should be called to change the status to "COMPLETED." If the action

resulted in an error, then the protected method `error(std::string errorMsg)` should be called to change the status to "ERROR" and update the error message.
When an action results in an error, the program should print to the screen:
`"Error: <error_message>"`

More details about the actions will be provided in section 3.4.

## 3.3 Ordering strategies

**Sweaty Customer** – This customer loves to sweat and always orders all the cardio activities from the beginning of the work_options to their end (as is received in the input file). (3-letter code – swt)

**Cheap Customer** – This is a customer that always orders the cheapest workout from the workout options. This customer orders only once. (3-letter code – chp)

**Heavy Muscle Customer** – This type of customer is all about the muscles. They only perform anaerobic exercises (don't try to talk them out of it!). They always order all the anaerobic exercises from the most expensive to the cheapest. (3-letter code – mcl)

**Full Body Customer**– This type of customer is all about the versatility of the workout. They always start with the cheapest cardio exercise offered, continue to the most expensive mix-type workout, and finish with the cheapest anaerobic exercise they can find. (3-letter code – fbd)

Notes:

- ~~If a customer cannot complete his order (for example – A heavy muscle customer tries to order, but the exercise options have no anaerobic workout), they won't order at all. (No need to print an error or log this operation).~~
- When the strategy is ordering the "most expensive" or "cheapest" exercise, and there is more than one such exercise, the customer will order the exercise with the smallest id.

## 3.4 Actions

Below is the list of all actions that the user can request. Each action should be implemented as a class derived from the class BaseAction.

- **OpenTrainer –** Opens a given trainer's workout session and assigns a list of customers (Customer object). If the trainer doesn't exist or the trainer's workout session is already open, this action should result in an error: "Workout session does not exist or is already open." Each customer has a unique id, as described earlier (Section 3.2), in addition to their name. A trainer can receive more than one customer with the same name.
    - Syntax:

```
open <trainer_id> <customer1>,< customer1_strategy> <customer2>,<customer2_ strategy>.
```

    - where the `<customer_strategy>` is the 3-letter code for the ordering strategy as described in section 3.3.
    - example:

"open 2 Shalom,swt Dan,chp Alice,mcl Bob,fbd"

Will open a workout session with trainer number 2. If trainer number 2 doesn't have an open session and has at least four available spots. If the customer list is bigger than the trainer capacity then you should insert the customers you can (by the order of the list) and the rest of them will be ignored and won't get an id.

You can assume that customer names consist of one word (without spaces).

- **Order –** When the customers enter the trainer's workout session, they can request different workout plans. This function starts the workout session by the order that was given to the trainer. After each open (OpenTrainer), you can assume that an order can occur only once. This function will perform an order from each customer in the trainer's group, and each customer will order according to his strategy. After finishing with the orders, a list of all orders should be printed. If the trainer doesn't exist or isn't open, this action should result in an error: "Trainer does not exist or is not open".
  - Syntax: `order <trainer_id>`
  - Example:
    `"order 2"` - Start the group exercise of trainer number 2, and then print:

```
Shalom Is Doing Zumba
Shalom Is Doing Rope Jumps
Dan Is Doing Rope Jumps
Alice Is Doing Pilates
Alice Is Doing Yoga
Bob Is Doing Rope Jumps
Bob Is Doing CrossFit
Bob Is Doing Yoga
```

- **MoveCustomer** – Moves a customer from one trainer to another. Also, it moves all orders made by this customer from the origin trainer's salary to the new trainer's bill salary. If the origin trainer has no customers left after this move, the program will close his session. If either the origin or destination trainer is closed or doesn't exist, or if no customer with the received id is at the origin trainer's session, or if the destination trainer has no available spots for additional customers, this action should result in an error: "Cannot move customer". This action can only occur after the workout session has started (OpenTrainer).
  - Syntax:
    `move <origin_trainer_id> <dest_trainer_id> <customer_id>`
  - Example:
    `"move 2 3 5"` will move customer 5 from trainer 2 to trainer 3.

- **Close** – Closes a given trainer session. Should print the salary of the trainer to the screen. After this action, the trainer can accept new customers and can have an open session. The trainer's salary is accumulated, meaning that after each session, the salary is accumulated for the same trainer. If the trainer doesn't exist or the session isn't open, this action should result in an error: `"Trainer does not exist or is not open"`.
  - Syntax: `close <trainer_id>`
  - Example:
    `"close 2"` closes trainer 2, and then prints:

```
"Trainer 2 closed. Salary 740NIS"
```

- **CloseAll** – Closes all workout sessions in the studio, and then closes the studio and exits. The salaries of all the trainers that were closed by that action should be printed sorted by the trainer id in increasing order. Note that if all workout sessions are closed in the studio, the action will just close the studio and exit. This action never results in an error.
    - Syntax: `closeall`
    - Example:

      `"closeall"`

      will print:

      ```
      "Trainer 2 closed. Salary 740NIS"
      "Trainer 4 closed. Salary 600NIS"
      "Trainer 5 closed. Salary 330NIS"
      ```

- **PrintWorkoutOptions** – Prints the available workout options of the studio. This action never results in an error.

  Each workout option in the studio should be printed in the following manner:

  `<workout_name> <workout_type> <workout_price>`

    - Syntax: `workout_options`
    - Example:
    - `"workout_options"` will print:

      ```
      Yoga, Anaerobic, 90
      Pilates, Anaerobic, 110
      Spinning, Mixed, 120
      Zumba, Cardio, 100
      Rope Jumps, Cardio, 70
      CrossFit, Mixed, 140
      ```

- **PrintTrainerStatus** – Prints a status report of a given trainer. This action can only occur after the workout session has started (OpenTrainer).

  The report should include:

    - The trainer status.
    - A list of customers that are participating in the trainer's workout session.
    - A list of orders done by each customer.
      This action never results in an error.
    - Syntax: `status <trainer_id>`
    - Output format:

      ```
      Trainer <trainer_id> status: <open/closed>
      ```

```
Customers:
<customer_1_id> <customer_1_name>
...
<customer_n_id> <customer_n_name>
Orders:
<workout_name> <workout_price> <customer_id>
...
<workout_name> <workout_price> <customer_id>
Current Trainer's Salary: <trainers_salary>
```

- Examples:
  - For trainer 2 (Trainer ID = 2), where Shalom is doing Zumba and Rope Jumps, Dan is doing Rope Jumps, Alice is doing Pilates and Yoga, Bob is doing Rope Jumps, CrossFit, and Yoga. the call `status 2` will print:

    ```
    Trainer 2 status: open
    Customers:
    0 Shalom
    1 Dan
    2 Alice
    3 Bob
    Orders:
    Zumba 100NIS 0
    Rope Jumps 70NIS 0
    Rope Jumps 70NIS 1
    Pilates 110NIS 2
    Yoga 90NIS 2
    Rope Jumps 70NIS 3
    CrossFit 140NIS 3
    Yoga 90NIS 3
    Current Trainer's Salary: 740NIS
    ```

  - For trainer 3 (trainer id = 3) which is closed, "status 3" will print:

    ```
    Trainer 3 status: closed
    ```

- **PrintActionsLog** – Prints all the actions that were performed by the user (excluding current log action), from the first action to the last action. This action never results in an error.
  - Syntax: `log`
  - Output format:

```
<action_n_name> <action_n_args> <action_n_status>
...
<action_1_name> <action_1_args> <action_1_status>
```

Where action's name is the action's syntax, action's args are the action's arguments. The status of each action should be "Completed" if the workout session order was completed successfully, or "Error: <error_message>" otherwise.

- Example:

  In case these are the actions that were performed since the studio was opened:

  ```
  open 2 John,chp Pete,mcl
  open 2 Maggie,swt - Trainer 2 workout session is already open!
  Error: Trainer does not exist or is already open
  ```

- Then the "log" action will print:

  ```
  open 2 John,chp Pete,mcl Completed
  open 2 Maggie,swt Error: Trainer does not exist or is already open
  ```

- **BackupStudio** – save all studio information (studio's status, trainers, orders, workout options, and actions history) in a global variable called "backup". The program can keep only one backup: If it's called multiple times, the latest studio's status will be stored and overwrite the previous one. This action never results in an error.
  - Syntax: `backup`
  - Instructions: To use a global variable in a file, you should use the reserved word
  - "extern" at the beginning of that file, e.g: `extern Studio* backup;`
- **RestoreStudio** – restore the backed-up studio status and overwrite the current studio status (including studio status, trainers, orders, workout plan, and actions history). If this action is called before backup action is called (which means "backup" is empty), then this action should result in an error: `"No backup available"` Syntax: `restore`

Note: in this assignment, we assume that the action's input provided by the user is following the above syntax (no need to perform input checks).

## 3.5 Input file format

The input file contains the arguments of the program, **each in a single line**, by the following order:

- Parameter 1: number of trainers in the studio.
- Parameter 2: a list of trainer's capacities (maximum spots in each trainer's workout session) separated by a comma:

  `<trainer 1 capacity>,<trainer 2 capacity> …`
- Parameter 3: a list of workout options, each workout option is in a separate line, including workout option name, workout type (Anaerobic, Mixed, and Cardio), and a price separated by a comma:
  `<workout option name>,<workout type>,<workout price>`

The order determines the workout ID is the order in which it appears in the workout options section of the config file. So, in the following example, Yoga's ID will be 0, Pilates 1,...

Lines starting with '#' are comments. Empty lines and comments should be ignored when parsing the file.

Example:

```
# Number of trainers
4

# Traines
6,6,8,4

# Work Options
Yoga, Anaerobic, 90
Pilates, Anaerobic, 110
Spinning, Mixed, 120
Zumba, Cardio, 100
Rope Jumps, Cardio, 70
CrossFit, Mixed, 140
```

## 4 Provided files

The following files will be provided for you on the assignment homepage:

- Studio.h
- Trainers.h
- Action.h
- Customer.h
- Workout.h
- Main.cpp

You are required to implement the supplied functions and to add the Rule-of-five functions as needed. All the functions declared in the provided headers must be implemented **correctly**, i.e., they should perform their appropriate purpose according to their name and signature. You are **NOT ALLOWED** to modify the signature (the declaration) of any of the supplied functions. We will use these functions to test your code. Therefore any attempt to change their declaration might result in a compilation error and a significant reduction of your grade. You also must not add any global variables to the program.
**Keep in mind that if a class has resources, ALL 5 rules have to be implemented even if you don't use them in your code. Do not add unnecessary Rule-of-five functions to classes that do not have resources.**

## 5 Examples

See the assignment page for input and output examples.

## 6 Submission

- Your submission should be in a single zip file called "student1ID-student2ID.zip". The files in the zip should be set in the following structure:
  - src/
  - include/
  - bin/
  - makefile

  **src/** directory includes all .cpp files that are used in the assignment.

  **Include/** directory includes the header (.h or *.hpp) files that are used in the assignment. **bin/** directory should be empty, no need to submit binary files. It will be used to place the compiled file when checking your work.

- The makefile should compile the CPP files into the bin/ folder and create an executable named "**studio**" and place it also in the bin/ folder.
- Your submission will be built (compile + link) by running the following commands: "make".
- Your submission will be tested by running your program with different scenarios, and different input files, for example, "bin/rest input_file1.txt "
- Your submission must compile without warnings or errors on the department computers.
- We will test your program using VALGRIND in order to ensure no memory leaks have occurred. We will use the following Valgrind command:
- `valgrind --leak-check=full --show-reachable=yes` [program-name] [program parameters].
  The expected Valgrind output is:

```
definitely lost: 0 bytes in 0 blocks indirectly lost: 0 bytes in 0 blocks
possibly lost: 0 bytes in 0 blocks suppressed: 0 bytes in 0 blocks
```

```
ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

We will ignore the following error only:

still reachable: 72,704 bytes in 1 blocks (known issue with std). **We will not ignore** "still reachable" with different values than **72,704** bytes in **1** blocks

- Compiler commands must include the following flags:

```
-g -Wall -std=c++11
```

## 7 Recommendations

1. Be sure to implement the rule of five as needed. We will check your code for correctness and performance.
2. After you submit your file to the submission system, re-download the file you have just submitted, extract the files and check that it compiles on the university labs. Failure to properly compile or run on the department's computers will result in a zero grade for the assignment.

בהצלחה!