Systems Programming – Assignment #1
Restaurant management System
2019/Fall

1 Before You Start

- It is mandatory to submit all of the assignments in pairs. It is recommended to find a partner as soon as possible and create a submission group in the submission system. Once the submission deadline has passed, it will not be possible to create submission groups even if you have an approved extension.
- 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 functions that are declared in them.

KEEP IN MIND

While you are free to develop your project on whatever environment you want, your project will be tested and graded ONLY on a CS LAB UNIX machine. It is your own responsibility to deliver a code that compiles, links and runs on it. Failure to do so will result in a grade 0 to 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, it is unpleasant for us, at least as it is for you, to fail your assignments, 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 this assignment you will write a C++ program that simulates a restaurant management system. The program will open the restaurant, assign customers to tables, make orders, provide bills to the tables, and other requests as described below.

The program will get a config file as an input, which includes all required information about the restaurant opening - number of tables, number of available seats in each table, and details about the dishes in the menu. The format of the input file is further described in part 3.5.

There are 4 types of customers in this restaurant, each customer type has its own way of ordering from the menu (an ordering strategy). An order may be taken from a table more than once, in such cases some customers may order a different dish. (Ordering strategies are described in part 3.3)

Each table in the restaurant has a limited amount of seats available (this info is provided in the config file). The restaurant can’t connect tables together, nor accommodates more customers than the number of seats available in a table. In this restaurant, it’s impossible to add new customers to an open table, but it’s possible to move customers from one table to another.

A bill of a table is the total price of all dishes ordered for that table.

3.1 The Program flow

The program receives the path of the config file as the first command line argument. Once the program starts, it opens the restaurant by calling the *start()* function, followed by printing “Restaurant 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

Restaurant – This class holds a list of tables, menu, and other information that is relevant to the restaurant.

Table – This class represents a table in the restaurant. Each table has a finite number of available seats (provided in the config file). It also holds a status flag that indicates whether the table is open, a list of orders done in this table, and a list of customers. Table ids starts from 0.

Dish – This class represents a dish in the menu. It has an id, name, price, and a type (described later).

Customer – This is an abstract class for the different customers classes. There are several types of customers, and each of them has a different ordering strategy. Each customer that arrives to the restaurant will get a number (id) that will serve as an identifier as long as he is seating in the restaurant. This number will be a serial number of all customers that arrived so far, starting from 0 (the first customer will get 0, second customer will get 1 , etc.). Note that this “id” serves as a temporary identifier- if a customer leaves the restaurant and then comes back, he will get a new id.

The class has a pure virtual method order(menu) which receives the menu, and returns a vector of dishes IDs that were ordered by the customers. Ordering strategies are further described in 3.3.

BaseAction – This is an abstract class for the different action classes. The class contains a pure virtual method *act(Restaurant& rest)* which receives a reference to the restaurant 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: “Pending” for actions that weren't performed yet, “Completed” for successfully completed actions, and “Error” for actions which couldn't be completed.

After each action is completed- if the action was completed successfully, the protected method *complete()* should be called in order to change the status to “COMPLETED”. If the action resulted in an error then the protected method *error(std::string errorMsg)* should be called, in order 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

Vegetarian Customer – This is a customer that always orders the vegetarian dish with the smallest id in the menu, and the most expensive beverage (Non-alcoholic). (3-letter code – veg)

Cheap Customer – This is a customer that always orders the cheapest dish in the menu. This customer orders only once. (3-letter code – chp)

Spicy Customer – This is a customer that orders the most expensive spicy dish in the menu. For further orders, he picks the cheapest non-alcoholic beverage in the menu. The order might be equal to previous orders. (3-letter code – spc)

Alcoholic Customer – This is a customer who only orders alcoholic beverages. He starts with ordering the cheapest one, and in each further order, he picks the next expensive alcoholic beverage. After reaching the most expensive alcoholic beverage, he won't order again. (3-letter code – alc)

Notes:

- If a customer cannot complete his order (for example – A vegetarian customer tries to order but the menu has no vegetarian dish), he won't order at all.
- When the strategy is ordering the “most expensive” or “cheapest” dish, and there is more than one such dish, then the dish with the smallest id will be ordered.

3.4 Actions

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

Open Table – Opens a given table and assigns a list of customers to it. If the table doesn't exist or is already open, this action should result in an error: “Table does not exist or is already open”.
Syntax: open <table_id> <customer_1>**,**< customer_1_strategy> <customer_2>**,**< customer_2_ strategy> ….
where the <customer_ strategy> is the 3-letter code for the ordering strategy as described in section 3.3.
Example:
"open 2 Shalom**,**veg Dan**,**chp Alice**,**veg Bob**,**spc" will open table number 2, and will seat the four customers in it, in case that table number 2 was not open and it has at least 4 available seats.
You can assume customers names consist of one word (without spaces).

- **Order –** Takes an order from a given table. This function will perform an order from each customer in the table, and each customer will order according to his strategy. After finishing with the orders, a list of all orders should be printed. If the table doesn't exist, or isn't open, this action should result in an error: “Table does not exist or is not open”.
  Syntax: order <table_id>
  Example:
  "order 2" – Takes an order from table number 2, and then prints:

```
Shalom ordered Salad
Shalom ordered Milkshake
Dan ordered Water
Alice ordered Chili con carne
```

- **Move customer** – Moves a customer from one table to another. Also moves all orders made by this customer from the bill of the origin table to the bill of the destination table. If the origin table has no customers left after this move, the program will close the origin table. If either the origin or destination table are closed or doesn't exist, or if no customer with the received id is in the origin table, or if the destination table has no available seats for additional customers, this action should result in an error: “Cannot move customer”.
  Syntax: move <origin_table_id> <dest_table_id> <customer_id>
  Example:
  "move 2 3 5" will move customer 5 from table 2 to table 3.

- **Close** – Closes a given table. Should print the bill of the table to the screen. After this action the table should be open for new customers. If the table doesn't exist, or isn't open, this action should result in an error: “Table does not exist or is not open”.
  Syntax: close <table_id>
  Example:
  "close 2" closes table 2, and then prints:

```
Table 2 was closed. Bill 500NIS
```

- **Close all** – Closes all tables in the restaurant, and then closes the restaurant and exits. The bills of all the tables that were closed by that action should be printed sorted by the table id in an increasing order. Note that if all tables are closed in the restaurant, the action will just close the restaurant and exit. This action never results in an error.
  Syntax: closeall
  Example:

"closeall" will print:

```
Table 2 was closed. Bill 500NIS
Table 4 was closed. Bill 200NIS
Table 5 was closed. Bill 600NIS
```

- **Print menu** – Prints the menu of the restaurant. This action never results in an error. Each dish in the menu should be printed in the following manner:
  <dish_name> <dish_type> <dish_price>
  Syntax: menu
  Example:
  "menu" will print:

```
Salad VEG 40NIS
Water BVG 20NIS
Chili con carne SPC 400NIS
```

- **Print table status** – Prints a status report of a given table. The report should include the table status, a list of customers that are seating in the table, and a list of orders done by each customer. If the table is closed, only the table status should be printed. This action never results in an error.
  Syntax: status <table_id>
  Syntax of the output:
  Table <table_id> status: <open/closed>
  Customers:
  <customer_1_id> <customer_1_name>
  …
  <customer_n_id> <customer_n_name>
  Orders:
  <dish_name> <dish_price> <customer_id>
  …

<dish_name> <dish_price> <customer_id>

Current Bill: <total_bill>

Examples:

Example 1: In table 2 (table id = 2), where Shalom ordered salad and water, Dan ordered salad, and Alice ordered Chili con carne, "status 2" will print:

```
Table 2 status: open
Customers:
0 Shalom
1 Dan
2 Alice
Orders:
Salad 40NIS 0
Water 20NIS 0
Salad 40NIS 1
Chili con carne 400NIS 2
Current Bill: 500NIS
```

Example 2: For table 3 (table id = 3) which is closed, "status 3" will print:

```
Table 3 status: closed
```

- **Print actions log** – 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

  Syntax of the output:

  <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 order was completed successfully, or "Error: <error_message>" otherwise.
Example:
In case these are the actions that were performed since the restaurant was opened:
open 2 Jon,spc Pete,veg
open 2 Roger,spc Keith,alc
Error: Table does not exist or is already open

Then the "log" action will print:

```
open 2 Jon,spc Pete,veg Completed
open 2 Roger,spc Keith,alc Error: Table does not exist or is already open
```

- **Backup restaurant** – save all restaurant information (restaurant's status, tables, orders, menu and actions history) in a global variable called "backup". The program can keep only one backup: If it's called multiple times, the latest restaurant's status will be stored and overwrite the previous one. This action never results in an error.
  Syntax: backup
  Instructions: in order to use a global variable in a file, you should use the reserved word "extern" at the beginning of that file, e.g: "extern Restaurant* backup;"

- **Restore restaurant** – restore the backed up restaurant status and overwrite the current restaurant status (including restaurant's status, tables, orders, menu 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 actions 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 tables in the restaurant.
- Parameter 2: a list of tables capacities (maximum seats available) separated by comma: <table 1 number of seats>,<table 2 number of seats> …
- Parameter 3: a list of dishes, each dish in a separate line, including dish name, dish type (3-letters code: Vegetarian – VEG, Beverage – BVG, Alcoholic – ALC, Spicy – SPC) and a price separated by comma:
  <dish name>**,**<dish type>**,**<dish price>

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

Example:

```
#Number of tables.
3

#Tables
4,10,10

#Menu
Beer,ALC,50
Salad,VEG,40
Water,BVG,10
Wine,ALC,60
Chili con carne,SPC,200
```

4 Provided files

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

Restaurant.h

Table.h

Action.h

Customer.h

Dish.h

Main.cpp

You are required to implement the supplied functions and to add the Rule-of-five functions as needed. All the functions that are declared in the provided headers must be implemented **correctly**, i.e. they should perform their appropriate purpose according to their name and their 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 major deduction 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 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 ”**rest**” and place it also in the bin/ folder.
- Your submission will be build (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 that 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!