Systems Programming – Assignment #1
Food Warehouse Management System
2024/Fall

TAs in charge: Nir Sorani, Shir Buchner

Lecturer: Meni Adler

1 Before You Start

- It is mandatory to submit all the assignments in pairs. If you can't find a partner, use the designated forum.
- Read the assignment together with your partner and make sure you understand the tasks. **Before writing any code or asking questions, make sure you read the whole assignment.**
- Skeleton files 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.
- For any request for ease or postponing the deadline for submitting, please contact the responsible lecturer.
- Note that we marked every class members/methods you will encounter in the skeleton files, and code examples in this color, to help you develop intuition while reading this.

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 a faraway Country, SPLand, A war is going on for few months.
As part of the efforts to contribute and take care of the country’s citizens, few citizens decided to establish a food warehouse that supplies others food packages.

The center is operated by volunteers in different roles, and their job is to handle orders made by the customers.

The warehouse thanks in advance all the volunteers who do their best to help and improve the morale of the country’s citizens.

In this assignment, you will write a C++ program that simulates this “Food Warehouse management system”. The program opens the warehouse, assigns volunteers and customers, assigns orders, handles them, performs steps in the simulation, and more actions that will be described later.

3.1 The Program Flow

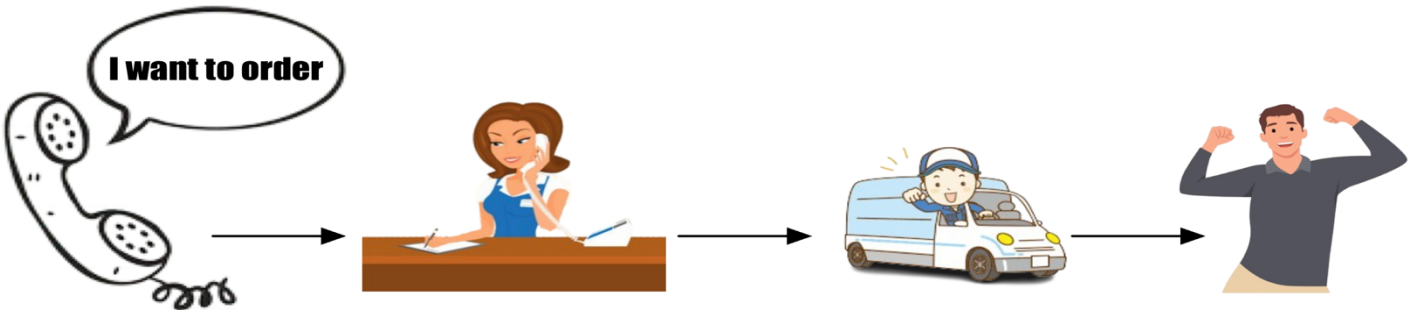
The program receives the path of the config file as the first command-line argument:

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

The config file describes the initial state of the warehouse before we start the simulation. It contains info of the initial customers and volunteers.
Once the program starts, it initializes the warehouse according to a given config file, and then starts the simulation by calling the `start()` method of Warehouse, followed by printing to console: `"Warehouse is open!"` .

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.

Generally, the warehouse order handling policy is as follows:



Step 1: Customer places an order. Step 2: Collector Volunteer answers the phone and collects the food. Step 3: Driver Volunteer delivers the food to the customer. Step 4: Happy customer

The actions in our program - placing an order, creating new customers, checking order status, performing simulation steps, etc. -, implemented by you, will help to manage the warehouse well.

****All actions and their uses will be specified in section 3.3.**

To solve this assignment, you will use the OOP approach. We will provide you skeleton files with some classes header files, and a main.cpp. You required to implement all the classes.

You are free to add more classes, members, and methods to the existing classes, but you **must not** change the given signatures.

3.2 Classes

Warehouse – This class holds a list (Vector) of volunteers, customers and different kinds of orders:

- Pending Orders – list of orders that waits to be processed.
- In Process Orders - list of orders that are currently being processed.
- Completed Orders – list of Orders that have been completed and reached their customers.

The warehouse also has customerCounter and volunteerCounter class that providing unique IDs, and isOpen which determines the warehouse status.

Customer - This class represents different types of customers. Each customer will get a unique ID that serves him during the simulation. This ID associates with any order he does during the simulation.
There are two types of customers:

- Soldier Customer
- Civilian Customer

Volunteer – This is an abstract class for the different volunteer classes. There are several types of volunteers, and each of them has a different role (Collector/Driver) which reflects in additional members and methods.

Note the class has a pure virtual method step(), which is responsible to make a step in the simulation (more details in section 3.3) according to the volunteer role and type.

There are 2 volunteer roles, each of them has 2 different types:

- **Collector** – This volunteer is responsible for orders that have just been received in the warehouse, entrusted with handling them including "picking up" the package and delivering them to the drivers. For the sake of simplicity, he has a coolDown member

that determines his processing time for each order (depends on the volunteer instance, not on the order).

- **Limited Collector** – This volunteer **extends** the "regular" collector. This volunteer has a limit to the total number of orders he can process - maxOrders. After reaching this limit, you should delete him.
- **Driver** – This volunteer is responsible for delivering the package to the customer. A driver can "process" and deliver only packages that have already been processed by some collector. The delivery time depends on the customer's distance from the warehouse. Each driver has a limit regarding the maxDistance, and therefore cannot handle orders whose distance is greater than this limit.
- **Limited Driver** – This volunteer **extends** the "regular" driver. This volunteer has a limit to the total number of orders he can process. After reaching this limit, you should delete him. This member called maxOrders.

***Reaching here, we recommend you take another look at the program flow diagram again (3.1), to be sure you understand how things should work.**

Order – This is a class who describes an order. Each order made by a customer and had its own status and unique ID. In our program, OrderStatus is an **enum** (explanation below) and could be one of the following:

- **PENDING** – when a customer places an order it should be stored in the warehouse until one of the collectors will handle it.
- **COLLECTING** – when one of the collectors has been associated with the order, the status of the order changes from pending to collecting.
- **DELIVERING** – when one of the drivers has been associated with the order, the status of the order changes from collecting to delivering.
- **COMPLETED** – when the customer gets his package (Driver finishes processing the order), the status of the order changes from delivering to completed.

enum - 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).

Each order associated with customer ID and the volunteers who handled it.

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(Warehouse &warehouse) which receives a reference to the warehouse 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" .
resulted in an error, then the protected method error(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_msg>"

We will emphasize and expand later with examples – Actions are the way to interact with the program, and it's kind of the engine of it. This is the way to advance the simulator few steps, to change its state by place new orders, create new customers and volunteers and so on.

***More details about the actions will be provided below**

3.3 Actions

SimulateStep – perform a step is a core action in the simulator. Performing one step in the simulation is actually moving one unit of time forward.

The scheme consists of 4 stages:

1. Go through all orders in pendingOrders list in warehouse and hand them over for the next operation depends on their status – Pending order will be passed to collector volunteer. Collecting order will be passed to driver. While a volunteer is processing an order, it should be in inProcessOrders.

   Notes:

   - Status of an order changes only after they are handled by new volunteers. For example, if a collector finished processing an order, the order status is still Collecting until it is associated

with a driver. This prevents a situation where an order status changes, while there is no volunteer available to complete the handling at the next stage of the order.

- For simplicity, you can associate the orders with the volunteers in any way you want, avoiding situations where the order is pending while there is an unemployed volunteer who can handle the order (driver cannot handle Pending order, only collector, so if all collectors are busy, it's fine). In addition, make sure there is no order starvation – when order waits to be processed while newer order in pendingOrders were already removed from the list. In other words, older orders should be associated with volunteer before any other newer order with the same volunteer role.

2. Perform a step in the simulation – decrease the timeLeft by one for each collector and decrease the distanceLeft of a driver by distancePerStep.
3. Go through all volunteers and check if they have finished with their order processing (collecting or delivery). Every volunteer that has finished his job will push the order to the pendingOrders or completedOrders lists in the warehouse. Update ordersLeft volunteer member in case he is Limited volunteer.
4. Each volunteer that reaches his maxOrders limit and finished handling his last order, should be deleted. We don't need him anymore. Furthermore, we don't want that in the next step we will associate any orders with him.

Note That:

distanceLeft cannot be a negative number, because it represents the distance to the customer. For example, if Moshe is a driver who delivers an order where his distanceLeft=4 and distancePerStep=6, so after one step in the simulation distanceLeft=0, even though it took less than one time unit.

- Syntax: step <number_of_steps>
- Example: step 3
- This action never results an error. Assume number_of_steps is positive number.

Order – This action creates an order. To create an order action, we only need the customer ID. After the order is created, it should be initialized with Pending status and pushed into

pendingOrders list in warehouse. At this stage, do not associate the order with any volunteer. It happens as part of the scheme we just define above in SimulateStep.

- Syntax: order <customer_id>
- Example: order 4
- This action should result an error if the provided customer ID doesn't exist, or the customer reaches his maxOrders limit: "Cannot place this order".

AddCustomer – This action creates a new customer and store him in warehouse.

- Syntax: customer <customer_name> <customer_type> <customer_distance> <max_orders>
- Example: customer Dani soldier 7 3
- This action never results in an error.

PrintOrderStatus – This action prints an information on a given order, includes its status, the volunteers that associated with it and the customer who place it.

- Syntax: orderStatus <order_id>
- Example: orderStatus 3 will prints:

  OrderId: 3
  OrderStatus: Pending/Collecting/Delivering/Completed // orderStatus enum
  CustomerId: <customer_id> // the actual customer ID
  Collector: <collector_id>/None //the actual collector ID or None in case the order didn't reach the Collecting stage
  Driver: <driver_id>/None //the actual driver Id or None in case the order didn't reach the Delivering stage.

- This action should result an error if the order_id doesn't exist: "Order doesn't exist".

PrintCustomerStatus – This action prints all information about a particular customer, includes his ID, his orders status, and how many orders he still may place. You may print the orders in any way you choose, and it's not necessary to print them in ascending order.

- Syntax: customerStatus <customer_id>

- Example: customerStatus 6 will prints:

  CustomerID: 6
  OrderID: <order_id>
  OrderStatus: Pending/Collecting/Delivering/Completed // status of the OrderID
  OrderID: <order_id> // second order
  OrderID: Pending/Collecting/Delivering/Completed // status of the second OrderID
  … // another Orders..
  numOrdersLeft: <num_orders_left> // Notice every customer have maxOrders member
- This action should result an error if the customer_id doesn't exist: "Customer doesn't exist".

PrintVolunteerStatus – This action prints all information about a particular volunteer, includes his ID, if he is busy or not, the order ID he currently processing, how much time is left until the processing of the order is completed, and how many orders he still may handle.

- Syntax: volunteerStatus <volunteer_id>
- Example: volunteerStatus 1 will prints:

  VolunteerID: 1
  isBusy: True/False (Depends if the volunteer handle something or not)
  OrderID: <order_id>/None (in case isBusy is False)
  timeLeft: <time_left/distance_left>/None (depends if the volunteer is a Collector/Driver/doesn't process any order)
  ordersLeft: <max_orders>/No Limit (depends on the type of the volunteer)
- This action should result an error if the volunteer_id doesn't exist: "Volunteer doesn't exist".

PrintActionsLog - Prints all the actions that were performed by the user (excluding current log action), from the first action to the last action.

- Input Syntax: log

- Output Format:

  `<action_1_name> <action_1_args> <action_1_status>`

  `<action_2_name> <action_2_args> <action_2_status>`

  `...`

  `<action_n_name> <action_n_args> <action_n_status>`

- Example:

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

  `customer Ben soldier 4 2` (Creating a soldier customer with distance=4 , maxOrders=2)

  `order 0` (Assume our warehouse gave Ben 0 as his unique ID)

  `order 0`

  `order 0`

  `simulateStep 1` (perform one step in the simulation)

  Then the “log” action will print:

  `customer Ben soldier 4 2 Completed`

  `order 1 Completed`

  `order 1 Completed`

  `order 1 Error: Cannot place this order` (Ben reached his maxOrders limit)

  `simulateStep 1 Completed`

- This action never results in an error.

Close – This action prints all orders with their status. Then, it closes the warehouse – changes its `isOpen` status, exits the loop and finishes the program. Make sure you free all memory before finishing the program, so you won’t have memory leaks.

- Syntax: `close`
- Output Format:

<order_1_id> <customer_id> <order_status>
<order_2_id> <customer_2_id> <order_2_status>
....
<order_n_name> <customer_n_id> <order_n_status>

- This action never results in an error.

BackupWarehouse - save all warehouse information (customers, volunteers, orders, and actions history) in a global variable called “backup”. The program can keep only one backup: If it's called multiple times, the latest warehouse’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 WareHouse* backup;
- This action never results in an error.

RestoreWarehouse - restore the backed-up warehouse status and overwrite the current warehouse status (warehouse itself, customers, volunteers, orders, and actions history).

- Syntax: restore
- 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".

3.4 Input file format

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

1. Customers – each line describes a customer in the following pattern:
   customer <customer_name> <customer_type> <customer_distance> <max_orders>
   For example:
   customer Maya soldier 4 2 //Maya is a soldier, dist=4 time units with maxOrders=2
   customer David civilian 3 1 //David is a civilian, dist=3 time units with maxOrders=1
2. Volunteers – each line describes a volunteer in the following pattern(read example):
   volunteer <volunteer_name> <volunteer_role>
   <volunteer_coolDown>/<volunteer_maxDistance> <distance_per_step>(for drivers only) <volunteer_maxOrders>(optional)

   For example:
   volunteer Noya collector 2 //"Regular" collector -there is no maxOrders.
   volunteer Ibrahim collector 3 2 //LimitedCollector with limitation of 2 orders.
   volunteer Din driver 13 4 2 //LimitedDriver with maxDistance=13,
   distPerStep=4,maxOrders=2
   volunteer Limor driver 8 3 //"Regular" driver with maxDistance=8,distPerStep=3 without maxOrders limit

   *Notice you parse correctly different roles, they might have different arguments which depends on their role, Ibrahim and Limor in this example emphasis the point.

4 Provided Files

The following files are provided for you in the skeleton.zip:

- WareHouse.h
- Order.h
- Customer.h
- Volunteer.h
- Action.h
- main.cpp

In addition, we provided for you ExampleInput.txt (config file), and RunningExmaple.pdf .

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 must 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

We attached to the assignment a file called **RunningExample** with few cases and their inputs/outputs.

For any other special scenario or mistake you detected, please post on the Assignment Forum.

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) 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 "warehouse" 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_file.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:
  All heap blocks were freed -- no leaks are possible
  ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

- Compiler commands must include the following flags:
  -g -Wall -Weffc++ -std=c++11 -Iinclude

7 Recommendation

- Be sure to implement Rule of 5 as needed. We will check your code for correctness and performance.
- After you submit your file to 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 a zero grade for the assignment.

GOOD LUCK AND ENJOY!