Systems Programming – Assignment #1
Reconstruction Simulation
2024/Fall

TAs in charge: Nir Sorani

Lecturer: Gil Einziger

1 Before You Start

- It is mandatory to submit all the assignments in pairs.
- 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 the distant land of SPLand, a long war has ravaged many areas, including cities and villages. Now, SPLand is focused on creating a comprehensive reconstruction plan for these regions. There is no one better suited than you to develop a simulation system aimed at evaluating various reconstruction plans, each distinguished by different prioritization strategies.

In this assignment, you will create a C++ program to manage and simulate various “reconstruction plans.” The program will define multiple plans, each focused on reconstructing specific settlement. It will also consider different strategies, such as whether to build a school or an office building, and perform simulation steps to evaluate how these decisions impact the outcome. Additional actions and details will be described later.

We are hopeful that presenting these results to SPLand’s government will lead to informed decisions, fostering a sense of security and prosperity for the residents of these settlements.

On this occasion, we would like to thank all IDF soldiers while risking their lives for our sake, and we wish for the return of the hostages home. It's not complete without them.

3.1 The Program Flow

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

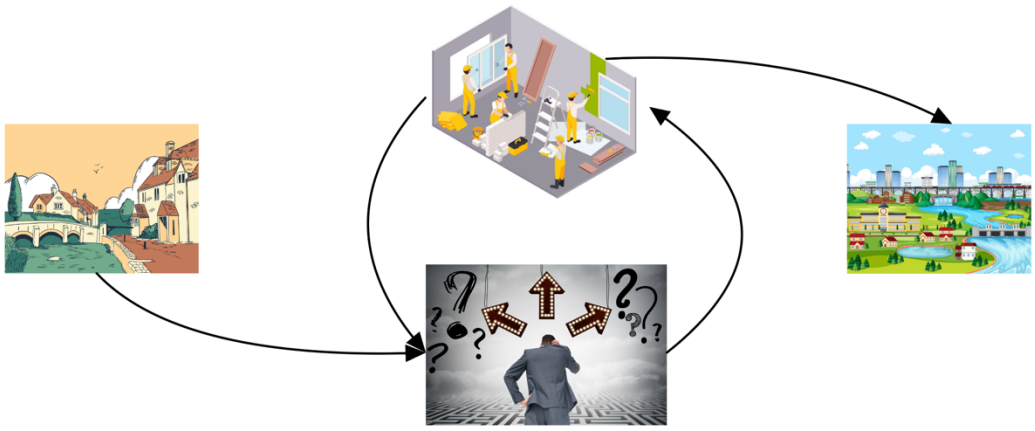
```
string configurationFile=argv[1]
```

The config file outlines the initial state of the settlements before the simulation begins. It also provides information about the objects that may be used during the simulation, such as their prices and their contributions to various indices (see Section 4 for more details).

Once the program starts, it initializes the simulation according to a given config file, and then starts the simulation by calling the `start()` method of the Simulation instance, followed by printing to console: `"The simulation has started"` .

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.

We can try to summarize the general flow using the following drawing:



Step 2 (main loop): Alternate between strategically selecting a facility to build and executing its construction and executes user pre-defined actions.

The actions in our program – create a plan, build a facility, checking facility/plan status, performing simulation steps, etc. They will be implemented by you and are the building blocks behind the simulation.

****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 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

Simulation – This class maintains a list of available settlements, plans, executed actions, and a collection of potential facilities specified in the configuration file. These facilities are considered for inclusion in the reconstruction plans.

It also includes a planCounter responsible for generating unique IDs, and an isRunning flag that indicates whether the simulation should be ended.

Settlement – This class describes a settlement. It has a unique name and one of the following SettlementType(s):

- Village – Allows to build only 1 facility at a time.
- City – Allows to build 2 facilities at a time.
- Metropolis – Allows to build 3 facilities at a time.

* *The number of facilities that can be built at a time is also called the "construction limit".*

FacilityType – This class represents a general facility, such as a school, park, office building, gym, or similar establishment. Each facility is identified by a unique name and has a price, measured in time units required to build it. While facilities belong to one of three categories (FacilityCategory enum), they can contribute to enhancing the score of all three following scores:

- **Life Quality score** – A score that is primarily influenced by facilities related to education, parks, and essential services for the settlement's citizens, such as medical centers.
- **Economy score** – A score that is primarily enhanced by facilities related to the economy, such as factories, office buildings, and roads.
- **Sustainability score** – A score primarily boosted by facilities focused on environmental protection, such as national parks, solar energy installations, and public transportation systems.

Facility - This class extends FacilityType by adding several additional fields:

- settlementName - Associates each facility instance with a specific settlement.
- FacilityStatus - Represents the status of the facility, initially – **UNDER_CONSTRUCTION**.

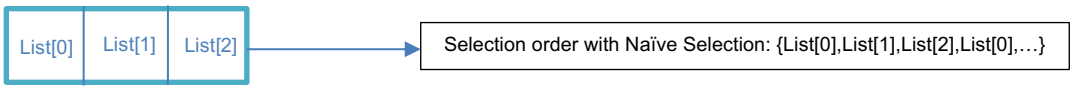
* We maintain both the FacilityType and Facility classes for distinct purposes:

  - FacilityType: Manages and stores the available facilities in a list, both after parsing and throughout the simulation, i.e. A more general representation of a facility.
  - Facility: Represents instances which associated with particular plans, enabling different instances to belong to different plans, or even in the same plan.

**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).

SelectionPolicy – This abstract class defines various selection policies through the selectFacility virtual method. It outlines different strategies for selecting the next facility to build. Derived classes should implement the selection logic within the - Facility& selectFacility(const vector<FacilityType>& facilitiesOptions) method. The selection policies available in our program are:

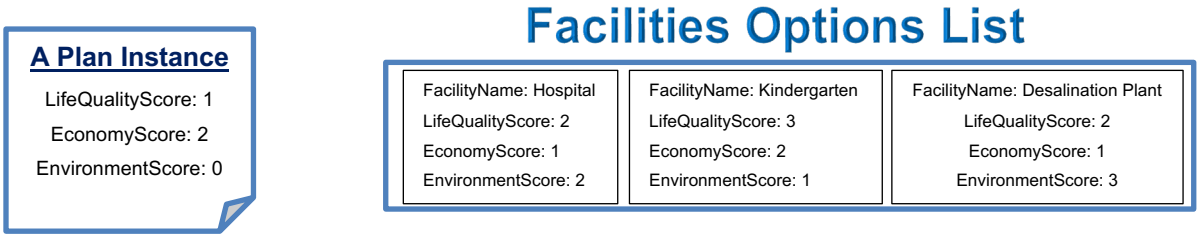
- Naïve Selection Policy – Selecting the next facility in the list. Done as follows:



  The class member lastSelectedIndex is maintained to track the last selection.
- Balanced Selection Policy – The selection of a facility depends on the plan scores, with the policy aiming to achieve the best balance among these scores. Practically, the chosen facility will be the one that minimizes the difference between the maximum and minimum scores.

  (An Example provided below)

- *For example, Assume the following scenario:*



Then, $distance('Hospital') = 1$,because after its construction the LifeQualityScore will be 3 and EconomyScore will be 2.
Similarly, $distance('Kindergarten') = 3$, $distance('DesalinationPlant') = 0$. Therefore 'Desalination Plant' will be chosen.

**In case of equality between different facilities, the policy will select the first one (lower index).*

- Economy Selection Policy – It prioritizes facilities in the economy category and selects the next one from this group.
  The class member lastSelectedIndex is maintained to track the last selection.
- Sustainability Selection Policy - It prioritizes facilities in the environment category and selects the next one from this group.
  The class member lastSelectedIndex is maintained to track the last selection.

* *In an event of a successful selection policy change (see Section 3.3), the lastSelectedIndex will be reset, except for the Balanced Selection Policy. For this policy, it will be initialized based on the plan scores as usual.*

Plan - This class represents a reconstruction plan; each assigned a unique ID for use throughout the simulation. Each plan is associated with a single settlement and oversees its reconstruction process.The class also includes a SelectionPolicy to determine the next facility to be constructed within the settlement, as part of the step() method detailed in Section 3.3.

* *A settlement may have multiple plans, allowing for the simultaneous consideration of various options. It maintains two lists: one for operational facilities and another for those under construction.*

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(Simulation &simulation) which receives a reference to the simulation 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 creating plans, changing selection policy and so on.

We want to emphasize once more - while parsing the configuration file and creating plans or facilities to prepare the simulation, you should not use actions at all. Actions are intended solely for interaction between the user and the program.

3.3 Actions

SimulateStep – Performing a step is a fundamental action in the simulation, representing the passage of one unit of time. During this process, the simulation iterates through all plans and executes their step() method according to the following scheme:

1. If the PlanStatus is BUSY ,then proceed directly to Stage 3. Otherwise, i.e. PlanStatus is AVALIABLE , move to the next stage.
2. Based on the selection policy, choose the next facility to build. If the construction limit hasn't been reached, continue selecting additional facilities to build, and repeat this process as needed.
   * *For example, at the start of the simulation, a city can build two facilities simultaneously, then select() method of the policy will be executed twice.*
3. Iterate over all underConstruction facilities and reduce their timeLeft by one. For any facility where timeLeft equals zero, you may change the FacilityStatus to OPERATIONAL, and move it to the facilities list.
   **This step may be implemented as part of Facility's step() method.*
4. Update the PlanStatus: If the length of the underConstruction list equals the construction limit, set the status to BUSY. Otherwise, set it to AVAILABLE.

* *Each iteration of the above scheme represents one time unit/step.*

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

AddPlan – This action creates a plan. To create a plan, we need the settlement for which it is created and a selection policy. The selection policies are abbreviated as follows:

- Naive Selection Policy – “nve”
- Balanced Selection Policy – “bal”
- Economy Selection Policy – “eco”
- Sustainability Selection Policy – “env”

Once the plan is created, initialize it with the AVAILABLE status and add it to the plans list in the simulation. At this stage, do not add any facilities; this will occur as part of the scheme defined earlier in SimulateStep.

- Syntax: plan <settlement_name> <selection_policy>
- Example: plan KfarSPL eco
- This action should result an error if the provided settlement name or selection policy doesn't exist: "Cannot create this plan".

AddSettlement – This action creates a new settlement. For this action we should get the settlement name and type, where the type is the integer value corresponds with the SettlementType enum class.

- Syntax: settlement <settlement_name> <settlement_type>(village/city/metropolis)
- Example: settlement KeremSPL 1(Associates with city)
- This action should result an error if the settlement name is already exist: "Settlement already exists".

AddFacility – This action creates and stores a new facility(actually FacilityType). The provided category integer value will be used to set the corresponding category enum value.

Syntax: facility <facility_name> <category> <price> <lifeq_impact> <eco_impact> <env_impact>

- Example: facility market 1(Associates with Economy) 5 3 5 1
- This action should result an error if the facility name is already exist: "Facility already exists".

PrintPlanStatus – This action prints an information on a requested plan, includes its status, selection policy, and facilities and their status.

- Syntax: planStatus <plan_id>
- Example: The output for planStatus 1 will be:

  PlanID: <plan_1_id>
  SettlementName <settlement_1_name>
  PlanStatus: BUSY/AVALIABLE // PlanStatus enum

SelectionPolicy: bal/eco/env

LifeQualityScore: <LifeQuality_score>

EconomyScore: < EconomyScore _score>

EnvrionmentScore: < EnvrionmentScore _score>

FacilityName: <facility_name>

FacilityStatus: UNDER_CONSTRUCTIONS/OPERATIONAL

FacilityName: <facility_name>

FacilityStatus: UNDER_CONSTRUCTIONS/OPERATIONAL

----- A list of all facilities associated with this plan -----

FacilityName: <facility_name>

FacilityStatus: UNDER_CONSTRUCTIONS/OPERATIONAL

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

ChangePlanPolicy – This action changes the selection policy of an existing plan. The selection policy specified in the command is the new policy.

- Syntax: changePolicy <plan_id> <selection_policy>
- Example: the output for changePolicy 3 nve will be:

  planID: 3

  previousPolicy: <previous_policy>

  newPolicy: Naive
- This action should result an error if the previous policy is the same as the desired one or if the planID doesn't exist: "Cannot change selection policy".

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 simulation began:

  plan Kfar_SPL env // Creates a plan for Kfar_SPL

  facility kindergarten 0 3 3 2 1

  changePolicy 1 env // Trying to Change policy for plan with id=1,Will print an error msg.

  changePolicy 1 bal

  step 2 // perform two steps in the simulation

  Then the “log” action will print:

  plan Kfar_SPL env COMPLETED

  facility kindergarten 0 3 3 2 1 COMPLETED

  changePolicy 1 env ERROR (Since the plan is already has env selection policy)

  changePolicy 1 bal COMPLETED

  step 2 COMPLETED
- This action never results in an error.
- The way you should spell each action as the <action_n_name> is the **Input Syntax** of the action. For example: SimulateStep - "**step** 5 COMPLETED".

Close – This action prints all plans along with their accumulated results, then terminates the simulation by updating its isRunning status, exiting the loop, and ending the program. Ensure that all allocated memory is freed before the program finishes to avoid any memory leaks.

- Syntax: close
- Output Format:

  PlanID: <plan_1_id>

  SettlementName <settlement_1_name> // The settlement corresponds with the plan

  LifeQuality_Score: <plan_1_QualityLife_score>

  Economy_Score: <order_1_Economy_score>

  Environment_Score: <order_1_ Environment _score>

  ….

  PlanID: <plan_N_id>

  SettlementName <settlement_N_name> // The settlement corresponds with the plan

  LifeQuality_Score: <plan_N_QualityLife_score>

  Economy_Score: <order_N_Economy_score>

  Environment _Score: <order_N_ Environment _score>
- This action never results in an error.

BackupSimulation - save all simulation information (plans, facilities, settlement, and actions history) in a global variable called “backup”. The program can keep only one backup: If it's called multiple times, the latest simulation’s snapshot will be stored and overwrite the previous one. This action never results in an error.

* *The backup action should be added to the actions log like any other action.*

- 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 Simulation* backup;
- This action never results in an error.

RestoreSimulation - restore the backed-up simulation snapshot and overwrite the current one (plans, facilities, settlements, and actions history).

* *The restore action will be added to the actions log after the restoration is complete, like any other action.*

- Syntax: `restore`
- If this action is called before backup action is called (which means "backup" is Nullptr), then this action should result in an error: `"No backup available"`.

4 Config file format

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

1. settlements – each line describes a settlement in the following pattern:
   `settlement <settlement_name> <settlement_type>`
   For example:
   `settlement Kfar_SPL 0` //Kfar_SPL is a village
   `settlement Kiryat_SPL 2` //Kiryat_SPL is a Metropolis
2. Facilities – each line describes a facilityType in the following pattern(read example):
   `facility <facility_name> <category> <price> <lifeq_impact> <eco_impact> <env_impact>`
   For example:
   `facility kindergarten 0 3 3 2 1` // Kindergarten – belongs to the life quality category, costs 3 time units, and contributes 3, 2, and 1 points respectively.
   `facility desalinationPlant 2 4 2 2 3` // Desalination plant – belongs to the environment category, costs 4 time units, and contributes 2, 2, and 3 points respectively.
3. Initial plans – each line describes a new plan for a settlement.
   `plan <settlement_name> <selection_policy>`
   For example:
   `plan Kiryat_SPL bal` // creates a plan for Kiryat_SPL, with a balanced selection policy.
   `plan Kfar_SPL eco` // creates a plan for Kfar_SPL, with an economy selection policy.

* Make sure you parse the settlements in the correct order, as it impacts the initialization of the plans.

** You may use the static method parseArguments() from the Auxiliary class. Additionally, you are free to implement any additional auxiliary methods as needed.

5 Provided Files

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

- Simulation.h
- Plan.h
- Settlement.h
- SelectionPolicy.h
- Facility.h
- Action.h
- Auxiliary.h
- Auxiliary.cpp
- main.cpp

In addition, we provided for you ExampleInput.txt (config file), and a (Short) Running Example.

You are required to implement the supplied functions and to add the Rule-of-five(After the fourth practical session; RO3 after the third practical session) 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.

6 Examples

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

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

7 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 "simulation" and place it also in the bin/ folder.
- Your submission will be built (compile + link) by running the following commands: "make".
  ***YOU MUST UNDERSTAND AND BE FAMILIAR WITH YOUR MAKEFILE!!!**
- 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
```

8 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!