Object Oriented Programming - 2023/2024

Project assignment

The practical OOP work consists of a C++ program. This program must:

- Follow object-oriented principles and practices;
- Be done in C++, correctly using the semantics of this language;
- Use the classes/libraries used in classes, where appropriate, or others from the C++ standard library;
- Other libraries must not be used without the prior consent of teachers;
- It must implement the functionalities of the theme referred to in this statement;
- An approach based on the mere collage of excerpts from other programs or examples is not permitted. All code presented must be demonstrably understood by whoever presents it and explained in defence, otherwise, it will not be counted.

Overview

Theme and general concepts.

The aim is to build a simulation system in C++ for a home controlled by interconnected automation components. The simulator includes areas of the home, which have properties such as temperature, light, etc., which are inspected by sensors, which provide the read values to rules that are managed by rule processors that determine what to do depending on the readings of the sensors. There are also devices in the zones, which affect the properties of the zone where they are located and react to commands issued by the rule processors. The components in the zones are equipped with autonomous action, reacting to the properties of the zone, information from sensors, the passage of time, etc.

The simulator includes the notion of the passage of time. This characteristic is important for the functioning of the components. For example, a motion detection sensor can cause the activation of a light bulb, which will remain on for X moments. The passage of time in the simulator is completely independent of the real-time measured by the computer, being described in abstract units of “instants” and advancing to the next instant upon user command.

User interaction

The simulator will present on the screen a schematic view of the home, including the zones and the various components within them, and will allow the user to interact with the components and understand the properties of each of the zones. The user will also be able to modify the configuration of the home and components and their interconnections, as well as carry out high-level operations such as: creating a new home, safeguarding rule processors, recovering them later, etc. The user interface must be above all functional and informative, with graphic beautification aspects being secondary. There must be an area of the screen dedicated to viewing the home, a second area dedicated to entering user commands, and a third area dedicated to presenting textual information about what is happening in the simulator and also responses to commands entered by the user. It will be necessary to control the positioning of the cursor to print characters in a specific line/column and a library will be available to assist in this task and in colour management.

Main concepts involved

Home. The home consists of a set of housing **zones**. Each zone is placed in a position on a grid (one zone per position), and some positions may be empty (no housing zone at all). The grid that supports the (housing) zones has a dynamic size decided at runtime, with minimum dimensions of 2x2 positions and a maximum of 4x4 positions. The home will simply be the set of areas on the grid. The housing zones do not need to be in contiguous positions. The zones never overlap: each one is in its own position on the grid. Visually (user interface), information from one zone must not hide information from another zone.

Zones of housing. This is a prominent element, as it is an area that is associated with environmental properties of interest to sensors and devices. The zone contains all the home automation components: the sensors, the rule processors, and the devices (explained later).

Properties of the environment. Environment properties are associated with a zone and manifest as a key-value pair, where the key is a string and the value is a numeric quantity. For example: (“temperature”, degrees Celsius), or (“light”, amount of lumens). The sensors and devices that interact with the properties query the property using the key, obtaining/modifying its value.

Each property has associated minimum and maximum value rules. For example, the minimum value that the temperature can have is the absolute value of -273 C (rounded), there is no maximum value. The light has a minimum value of 0 lumens, with no maximum. Other properties may not have a minimum value but have a maximum, have neither a minimum nor a maximum, or have both. Each property will know how to manage its allowable values.

The simulator will support the following set of properties, but the code must be compatible with the addition of new properties, and this addition must be trivial (not require changing the entire code).

Properties	Unit	Minimum	Maximum
Temperature	degrees Celsius	-273	-
Light	Lumens	0	-
Radiation	Becquerel	0	-
Vibration (for movement)	Hertz (in the atmosphere)	0	-
Humidity (Rel.)	%	0	100
Smoke	Obscuration (%)	0	100
Sound	Decibels (in the audible range)	0	-

Sensors. Sensors are small devices that measure a certain property of the environment in the area in which they are located, providing this reading to one or more rules that are linked to it. Each sensor only deals with one property, being able to measure the value of that property. The sensor is completely passive and does not “feel” or react to the passage of time.

The simulator will allow, from the outset, the following sensors, and can be increased with new sensors (and in this case without the need to significantly modify the code).

Sensor	Character (for viewing)	Observed property
temperature	t	Temperature
mo**v**ement	v	Vibration
lu**m**inosity	m	Light
ra**d**iation	d	Radiation
humidity	h	Humidity
s**o**und	o	Sound
smoke	s	Smoke

Rule Processors. Rule processors (just “processor” from here on out) are devices that have rules. There is no direct connection between processors and sensors but between rules and sensors. The processors output a command (configurable). This command is communicated to one or more devices that are connected to the processor output when all existing rules are met. Each processor rule can be linked to a different sensor, so the activation of the processor command can be dependent on more than one sensor.

Rules of the processor. A rule is linked to a sensor (two different rules can be linked to the same sensor). Each rule implements a certain comparison condition between the value of the sensor to which it is linked and one or more configurable values. The result of the rules will simply be a logical value. This makes the processor's operation extremely simple: it has a set of rules, and at every moment it evaluates the result of each of them. If they are all true, the processor activates the configured command.

The rule processor is configurable as follows: rules can be added or removed, and the command to trigger is defined when building the rule processor and can later be changed.

There are several possible rules, and the simulator must already be prepared for the following:

Rule	Operation: “triggers command if the reading value…”	Parameters
equal_to	is equal to **x**	x
less_than	is less than **x**	x
greater_than	is greater than **x**	x
in_between	is between **x** and **y**	x, y
outside	is not between **x** and **y**	x, y

By default, a rule processor does not have any associated rules, and therefore never triggers its command. After adding a rule processor to a zone, it will need to configure itself by adding certain rules to it. The rules of a processor are not shared with any other processor.

The processor can implement arbitrarily complex conditions if that is the user's desire (it does not mean that they will be required). To understand the potential of this, and merely as an example, consider a processor with these two rules: between 10 and 40 for a given humidity sensor, and outside 20 to 30 for the same sensor. Basically, the processor command with these two rules is triggered if the observed value is between 10 and 20 or between 30 and 40 (that is, the effect of a logical “or” has been achieved). This complexity emerges from the set of rules and not from the program code, being a matter for the user who configures the simulator, and the programmer only has to worry about providing the functionality of the simple and elementary rules that are listed above and ensuring that the program it is expandable with new rules and new conditions if the need arises.

Devices. The devices can be controlled either by the user or by command of a rule processor to which they are connected. Devices can modify one or more properties of the zone in which they are located.

There are the following devices:

Device (name)	Character	Recognized commands	Effect: “If turned on…”	Effect: “If turned off”
he**a**ter	a	**on**- turn on the device **off**- turn off the device	Adds one degree Celsius to the temperature of the zone every 3 moments, up to a maximum of 50º, and adds 5 db of noise once.	removes 5 db of noise
sprinkler	s	**on**: turns on the device, remaining on for up to 5 instants after the off command **off**: turns off the device, but it continues to take effect as if it had been on for another 5 instants (remains of water still in the pipe)	At the first instant of switching on (i.e., once per period in which it is switched on): - Adds 50% relative humidity, up to a maximum of 75% humidity. - Adds 100 Hz vibration. Sets the smoke to 0 once in the second instant	removes 100 Hz of vibration
coole**r**	r	**on**- turn on the device **off**- turn off the device	It removes one degree Celsius from the zone temperature every 3 instants and adds 20 db of noise the first instant on.	removes 20 db of noise
lamp	l	**on**- turn on the device **off**- turn off the device	Adds 900 lumens once per instant it is on.	removes 900 lumens

Notes:

- Coincidentally, all the devices in the statement only recognize the “on” and “off” commands. However, you must prepare your program so that the devices (or any other devices) accept any commands, which will always be a string.
- Note that “when switching off” means that the effect only happens the first moment it is switched off: to have the switching off effect again it will have to be switched on again and then switched off.
- Some devices sense the passage of time and may need an internal instant counter and other flags to know when they take effect.

Each element of the simulator must be easily identified, and for this purpose, there must be a unique, incremental, and different code for each one, which allows the creation, deletion and association of the simulator components to be specified. This identifier is composed of a letter identifying the type of component ('d' - device, 's' - sensor, 'p' - processor, 'r' - rule, stored as lowercase) and an always increasing integer value, unique to home level. This ID should be generated as automatically as possible. For presentation purposes, in the case of devices, the letter will be shown as lowercase if the device is turned off and capitalized if it is on. Each zone of the house must also have its own identifier. These IDs will be essential for commands written by the user.

In certain cases, it may be necessary to identify which zone each piece of equipment belongs to, making it necessary to define an equipment identification mechanism, per zone.

Simulator operation

The entire simulation takes place in a simulated time in “instants”. When passing from one instant to the next, the simulator makes the rule processors analyse their rules, which consult the sensors, to determine which command should be issued. At every instant it also makes the devices perform their actions.

It is only advanced to the next instant at the user's command.

The user is free to perform whatever actions he or she wishes before moving on to the next instant. The visible consequences of your commands must be immediately represented.

User interface

At each new instant, the simulator provides information relating to the generality of what is happening: commands issued by the rule processors, devices that were turned on or off, properties that were changed (which one, from which zone and what is the new value), the number the instant you leave, etc.

The player's orders are given textually, according to the paradigm of written commands. Each order is written as a sentence in which the first word is a command and the following words are parameters or additional information. The line of text corresponding to the order is written in full, only then being interpreted and executed by the simulator. The program must validate the syntax and coherence of the command (have all the necessary information/parameters been written? Are the values that were supposed to be integers actually integers? Are they in the right order?). It can be assumed that everything will always be written in lowercase.

The screen should be organized into three logical areas. Each is dedicated to:

- presentation of a schematic view of the home (i.e., the grid with the zones, each with its components).
- introduction of user-written commands.
- presentation of results of written commands and information about what is happening in the simulator. The latter includes data such as properties that have their value changed, and devices that are turned on or off.

The details of the exact positioning of visual elements remain open as long as the informative capacity and quality of use are respected. Attention should be paid to the following restrictions or requirements:

- There will be no space on the screen to display menus and their use is prohibited.
- In each zone, there are properties, sensors, rule processors and devices. Sensors and devices must have a visual representation that is the character indicated in the previous tables, where they were described. The positioning of these characters in the screen space reserved for the zone is not important as long as they are visible. The zone only takes up “space” on the screen, and its representation in memory has nothing to do with character grids (again so that it is clear: the internal implementation of the zone has nothing to do with character grids).
- The visual representation of each zone also indicates the zone ID.
- The entire grid with the zones is intended to be displayed on the screen in order to avoid having to scroll through the information. It is necessary to plan how many characters in height and width are reserved for each zone, so as not to take up too much space, but at the same time ensure that you can visualize all the components in each zone.

User commands

The user can interact with the simulator and components through commands written in the area of the screen reserved for this purpose. The commands are described below.

In this description:

- <xxx> refers to a value (number or text), and the characters < and > are not exactly part of what should be written.
- < x | y | z> indicates that only one of x, y, z should be indicated (again, without the <>).
- [v] indicates that the v parameter is optional, and the characters [and] should not be written.

The results of the commands must be described in the area of the screen reserved for this purpose.

>> Commands for simulated time

next

Advance 1 instant, triggering the effects of components that react to the passage of time.

advance <n>

It advances **n** instants, one at a time, in each 1 the effect of the components that react to the passage of time is triggered.

>> Commands to manage home and zones

hnew <num rows> <num columns>

Creates a new home (zone grid) with the indicated size. There is no “default” home, so this command must be the first to be executed in a simulation. When creating the new home, the existing one will be destroyed as well as everything associated with it.

hrem

Removes home: deletes all zones and their contents and the grid itself (zones). Basically, it deletes all simulator content.

znew <row> <column>

Creates a new zone in the home grid at the indicated row, column position. The indicated position must be empty.

zrem <zone_ID>

Deletes the zone with the indicated ID and all its contents.

zlist

List all zones of the home. For each zone, it displays the ID and number of sensors, the number of processors, and the number of devices in it.

>> Commands to manage zones and their content

zcomp <zone_ID>

Lists the components existing in the zone with the indicated ID. Each component must include the type (sensor, processor or device), through a character “s, p or a”, followed by its numerical ID, followed by its name (heater, lamp, processor, etc.), and its state (if it is a sensor, the current reading value, if it is a device, the last command received, a processor only indicates the number of rules it has).

zprops <zone_ID>

Lists the properties recognized by the simulator in the zone with the indicated ID. Each property is described by its name and its current value.

pmod <zone_ID> <name> <value>

Modifies the value of the property whose name is given (if such a property exists) in the zone with the given ID (if it exists) to the specified value (if acceptable). The command reports whether the operation went well (just like all other commands).

cnew <zone_ID> <s | p | d> <type | command>

Adds a sensor(s) / processor(s) / device(s) of the indicated type to the zone that has the indicated ID. “type” refers to the type of sensor (what sensor) or device (which device). In the case of processor, there is only one type of processor and the word that is written in place of the type means the command to send when the rules are all evaluated as true (the command is anything that the devices that are connected to the processor output can understand, but generally it will be either “on” or “off” (they may be different if device classes are implemented that recognize them, e.g. increasing the volume of a television). The component ID is determined by the simulator and is indicated in the result of this command. The user must remember this ID as it will be necessary to interact with this component later. The sensor/device “types” must be those used in the respective tables indicated previously and must be recognized when written in lowercase. Use names consisting of a single word.

crem <zone_ID> <s | p | d> <ID>

Removes the component with the given ID from the specified zone. Since component IDs are unique, all you had to do was indicate the component ID and the program would search all zones. To simplify the program, the zone_ID is also indicated.

Important: if the component is being referred to by another component (for example, a sensor that is used by one or more rules), the coherence of the action must be guaranteed: either refuse to remove the component, or delete it “in cascade” all the components that use it (and then those that use the latter and so on), or somehow the component that uses it comes to know that it can no longer count on the component that was deleted. Either way, the simulator has to remain coherent.

>> Commands for rule processors

Note: In these commands, the zone ID is always specified so that the program can be simplified by searching for the components involved directly in the indicated zone.

rnew <zone_ID> <rules_processor_ID> <rule> <sensor_ID> [param1] [param2] [...]

Create a new rule of type “rule” (see rules table) and add it to the processor whose ID is indicated. The rule is associated with the sensor identified by sensor_ID. Depending on the rule that was created, one or more parameters may be required, which are indicated at the end of the command. Displays the ID that was generated for the rule. The rules have their own ID, independent of all other elements of the simulator and this is a growing number that is automatically managed.

pchange <zone_ID> <rules_processor_ID> <new command>

Changes the command of the indicated processor, in the specified zone. The command is one word.

rlist <zone_ID> <rules_processor_ID>

Lists the rules of the rules processor, one per line, indicating their name, their ID and the name and ID of the associated sensor.

rrem <zone_ID> <rules_processor_ID> <rule_ID>

Removes indicated rule from the specified rule processor.

Note: There are no commands to modify rules. This functionality is achieved by removing a rule and adding it back with the desired difference.

asoc <zone_ID> <rules_processor_ID> <device_ID>

Establishes the association between the output of the indicated processor and the specified device.

disa <zone_ID> <rules_processor_ID> <device_ID>

Removes the association between (disassociate) the output of the indicated processor and the specified device.

>> Commands to interact with devices

dcom <zone_ID> <device_ID> <command>

“Manually” sends the indicated command to the device whose ID was specified.

>> Commands for copying/retrieving rule processors

psave <zone_ID> <rules_processor_ID> <name>

Saves the state of a rule processor and everything associated with it to memory. A copy of everything that belongs to the processor is stored in this memory, and what does not belong is not stored (example: the rules belong, the sensors do not). The stored information includes the zone ID so you will later know where the processor came from. The copy of the processor is associated with a name that is later used to replace the processor. The name must be unique.

prestore <name>

Restores the processor previously saved in memory with the indicated name. The processor is returned to the zone it was in initially unless that zone has been deleted. The currently existing processor with that ID, if it still exists, is replaced by the restored copy (if it no longer exists, it works as a kind of adding processor). It may be necessary to check and adjust the contents of the restored processor to the new reality of the zone as it may have changed since the processor was saved.

prem <name>

Deletes a copy of the rules processor stored in memory.

plist

Displays a list of rule processor copies saved in memory. The list presented must include the name, the processor ID and the ID of the zonefrom which it came.

Note: The functionality of saving/recovering processor copies must be carefully thought out. There may be several alternatives: stay in class and in contact with teachers.

>> Additional general commands

exec <file name>

Loads a text file with commands and executes them. The file has one command per line. The commands must be executed as if they were entered on the keyboard by the user, that is, they have the same format and syntax as the commands indicated in this section of the statement (in fact, a command in a file can even load commands from another file). This command is essential as it allows you to have several commands previously saved in a file, greatly speeding up the program testing process.

exit

Close the program. This implies a “clean” exit, that is, releasing all allocated resources.

Restrictions on implementation

- The program should compile without any warnings or errors. The program must run without any errors or exceptions. The code must be robust and complete.
- An object-oriented program in C++ is expected. A solution not using object concepts (e.g. in C logic) will have 0 or very close to it.

There is more than one possible implementation strategy.

About the statement

- It is inevitably long given that a) it is necessary to describe characteristics of the simulator elements, b) it tries to answer in advance questions that normally arise. The length of the text does not necessarily translate into complexity or additional work.
- Some necessary classes for the most salient entities are deduced from the text, but other classes may be necessary.
- Some subjects are intentionally omitted. The statement acts as a representative situation of industry scenarios in which there are details that must be deduced and completed by whoever

performs the work. These aspects must be addressed according to common sense and without removing dimension, matter or complexity from the statement (in case of doubt, it is advisable to ask the teacher in good time). It will not be acceptable not to implement something whose need is obvious just because it was not explicitly requested in the statement. These open aspects include:

- Features whose need becomes obvious by others that are mentioned;
- Implementation aspects that involve the correct use of C++ mechanisms and semantics;
- Organizational aspects according to good programming practices and the object-oriented paradigm.

General work rules

Those described in the FUC and in the slides of the first theoretical class:

- Groups of two students maximum. Groups of 3 are not accepted. Groups of 1 must be previously justified with the teacher.
- For both goals, deliver a CLion project and a report in a zip file with the name indicated below.
- Mandatory defence in both goals. Both students must attend at the same time and whoever is absent will be left without a grade.
- Defense greatly affects the grade, it is individual in nature and students in the same group can have different grades.

Goals and deliverables

Goal 1 –November 25th

Requirements - features for goal 1:

- Reading commands and their validation (even if they don't do anything yet, they must be validated in terms of syntax and parameters).
- Reading the command file.
- Partial construction of the user interface. This is intended to ensure that students work with the library (provided) in a timely manner to handle the console.
- Construction of classes (still empty) for each of the generic types of components and a generic property. They can only be specified by .h.
- Class construction for zones.
- Construction of classes for the Processor and a generic rule (later improved).
- The project must already be properly organized in .h and .cpp.

Report– In this goal, the report is simplified, but must include a description of the options taken and a description of the structures used. It should also give an indication of the structuring of the work in terms of classes (which ones, what they are for / what they represent and what is the relationship between them).

-> File to be delivered: zip file (not rar or arj - it's zip) with the following mandatory name:

poo_2324_m**1**_name1_number1_name2_number2.zip

Goal 2 –December 30th

Requirements: complete program, with a detailed and complete report. The report must include a list of requirements implemented, partially implemented and not implemented (with an indication of the reason for those not implemented).
-> Zip file with name: poo_2324_m**2**_name1_number1_name2_number2.zip

In both deliveries:

- **Only one of the students** of the group makes the submission andn**necessarily associate the submission to the groupmate.**
- **Students must be enrolled in a class (they can be different classes in the same group).**